


配套97集（30小时）免费视频教程，让您轻松、快速用Python

Python3入门必备，小甲鱼手把手教您Python开发，

从真实案例中领略Python的真正魅力




清華大學出版社

水木书荟




清華大學出版社
北京

**内 容 简 介**

本书适合学习 Python3 的入门读者，也适用对编程一无所知，但渴望用编程改变世界的朋友们！本书提倡理解为主，应用为王。因此，只要有可能，小甲鱼（作者）都会通过生动的实例来让大家理解概念。

虽然这是一本入门书籍，但本书的“野心”可并不止于“初级水平”的教学。本书前半部分是基础的语法特性讲解，后半部分围绕着 Python3 在爬虫、Tkinter 和游戏开发等实例上的应用。

编程知识深似海，小甲鱼没办法仅通过一本书将所有的知识都灌输给你，但能够做到的是培养你对编程的兴趣，提高你编写代码的水平，以及锻炼你的自学能力。最后，本书贯彻的核心理念是：实用、好玩，还有参与。微信扫描书中对应二维码，亦可观看相关视频。



**图书在版编目（CIP）数据**

零基础入门学习 Python/小甲鱼编著. —北京：清华大学出版社，2016（2018.1 重印）
（水木书荟）
ISBN 978-7-302-43820-5

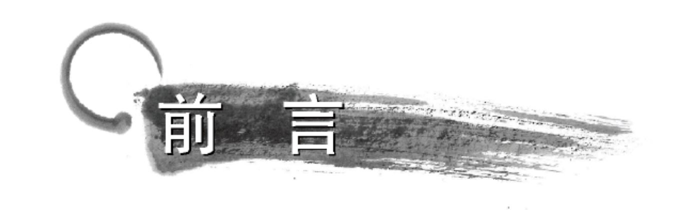
Ⅰ. ①零…　Ⅱ. ①小…　Ⅲ. ①软件工具－程序设计　Ⅳ. ①TP311.56

中国版本图书馆 CIP 数据核字（2016）第 101697 号

**责任编辑**：刘　星　李　晔
**封面设计**：刘　键
**责任校对**：时翠兰
**责任印制**：李红英

**出版发行**：清华大学出版社
**网　　址**：http://www.tup.com.cn，http://www.wqbook.com
**地　　址**：北京清华大学学研大厦 A 座　　**邮　　编**：100084
**社 总 机**：010-62770175　　**邮　　购**：010-62786544
**投稿与读者服务**：010-62776969，c-service@tup.tsinghua.edu.cn
**质量反馈**：010-62772015，zhiliang@tup.tsinghua.edu.cn
**课件下载**：http://www.tup.com.cn，010-62795954
**印 装 者**：清华大学印刷厂
**经　　销**：全国新华书店
**开　　本**：185mm×260mm　　**印　　张**：23　　**字　　数**：584 千字
**版　　次**：2016 年 10 月第 1 版　　**印　　次**：2018 年 1 月第8 次印刷
**印　　数**：49501～69500
**定　　价**：49.5 元

产品编号：067726-03



# 前 言

Life is short. You need Python.

——Bruce Eckel

上边这句话是 Python 社区的名言，翻译过来就是“人生苦短，我用 Python”。

我和 Python 结缘于一次服务器的调试，从此便一发不可收拾。我从来没有遇到一门编程语言可以如此干净、简洁，如果你有处女座情节，你一定会爱上这门语言。使用 Python，可以说是很难写出丑陋的代码。我从来没想过一门编程语言可以如此简单，它太适合零基础的朋友踏入编程的大门了，如果我有一个八岁的孩子，我一定会毫不犹豫地使用 Python 引导他学习编程，因为面对它，永远不缺乏乐趣。

Python 虽然简单，其设计却十分严谨。尽管 Python 可能没有 C 或 C++这类编译型语言运行速度那么快，但是 C 和 C++需要你无时无刻地关注数据类型、内存溢出、边界检查等问题。而 Python，它就像一个贴心的仆人，私底下为你都一一处理好，从来不用你操心这些，这让你可以将全部心思放在程序的设计逻辑之上。

有人说，完成相同的一个任务，使用汇编语言需要 1000 行代码，使用 C 语言需要 500 行，使用 Java 只需要 100 行，而使用 Python，可能只要 20 行就可以了。这就是 Python，使用它来编程，你可以节约大量编写代码的时间。

既然 Python 如此简单，会不会学了之后没什么实际作用呢？事实上你并不用担心这个问题，因为 Python 可以说是一门“万金油”语言，在 Web 应用开发、系统网络运维、科学与数字计算、3D 游戏开发、图形界面开发、网络编程中都有它的身影。目前越来越多的 IT 企业，在招聘栏中都有“精通 Python 语言优先考虑”的字样。另外，就连 Google 都在大规模使用 Python。

好了，我知道过多的溢美之词反而会使大家反感，所以我必须就此打住，剩下的就留给大家自己体验吧。

接下来简单地介绍一下这本书。一年前，出版社的编辑老师无意间看到了我的一个同名的教学视频，建议我以类似的风格撰写一本书。当时我是受宠若惊的，也很兴奋。刚开始写作就遇到了不小的困难——如何将视频中口语化的描述转变为文字。当然，我希望尽可能地保留原有的幽默和风趣——毕竟学习是要快乐的。这确实需要花不少时间去修改，但我觉得这是值得的。

本书不假设你拥有任何一方面的编程基础，所以本书不但适合有一定编程基础，想学习 Python3 的读者，也适合此前对编程一无所知，但渴望用编程改变世界的朋友！本书提倡理解为主，应用为王。因此，只要有可能，我都会通过生动的实例来让大家理解概念。虽然这是一本入门书籍，但本书的“野心”可并不止于“初级水平”的教学。本书前半部分是基础的语法特性讲解，后半部分围绕着 Python3 在爬虫、Tkinter 和游戏开发等实例上的应用。编程知识深似海，没办法仅通过一本书将所有的知识都灌输给你，但我能够做到的是培养你对编程的兴

趣，提高你编写代码的水平，以及锻炼你的自学能力。最后，本书贯彻的核心理念是：实用、好玩，还有参与。

本书对应的系列视频教程，可以在 http://blog.fishc.com/category/python 下载得到，也可扫描以下二维码关注微信号进行观看。



微信扫描书中对应二维码，亦可观看相关视频。

编者

2016 年 7 月

# 目 录


**第 1 章 就这么愉快地开始吧** ······ 1

1.1 获得 Python ······ 1
1.2 从 IDLE 启动 Python ······ 2
1.3 失败的尝试 ······ 3
1.4 尝试点儿新的东西 ······ 3
1.5 为什么会这样 ······ 4

**第 2 章 用 Python 设计第一个游戏** ······ 5

2.1 第一个小游戏 ······ 5
2.2 缩进 ······ 6
2.3 BIF ······ 6

**第 3 章 成为高手前必须知道的一些基础知识** ······ 8

3.1 变量 ······ 8
3.2 字符串 ······ 9
3.3 原始字符串 ······ 10
3.4 长字符串 ······ 11
3.5 改进我们的小游戏 ······ 12
3.6 条件分支 ······ 12
3.7 while 循环 ······ 13
3.8 引入外援 ······ 14
3.9 闲聊数据类型 ······ 15
3.9.1 整型 ······ 15
3.9.2 浮点型 ······ 16
3.9.3 布尔类型 ······ 16
3.9.4 类型转换 ······ 16
3.9.5 获得关于类型的信息 ······ 17
3.10 常用操作符 ······ 18
3.10.1 算术操作符 ······ 18
3.10.2 优先级问题 ······ 19
3.10.3 比较操作符 ······ 19
3.10.4 逻辑操作符 ······ 20

**第 4 章　了不起的分支和循环** …… 21

4.1　分支和循环 …… 21
4.2　课堂小练习 …… 23
4.3　结果分析 …… 24
4.4　Python 可以有效避免“悬挂 else” …… 24
4.5　条件表达式(三元操作符) …… 24
4.6　断言 …… 25
4.7　while 循环语句 …… 25
4.8　for 循环语句 …… 26
4.9　range() …… 26
4.10　break 语句 …… 27
4.11　continue 语句 …… 27

**第 5 章　列表、元组和字符串** …… 29

5.1　列表：一个“打了激素”的数组 …… 29
5.1.1　创建列表 …… 29
5.1.2　向列表添加元素 …… 29
5.1.3　从列表中获取元素 …… 31
5.1.4　从列表删除元素 …… 31
5.1.5　列表分片 …… 32
5.1.6　列表分片的进阶玩法 …… 33
5.1.7　一些常用操作符 …… 33
5.1.8　列表的小伙伴们 …… 35
5.1.9　关于分片“拷贝”概念的补充 …… 36
5.2　元组：戴上了枷锁的列表 …… 37
5.2.1　创建和访问一个元组 …… 37
5.2.2　更新和删除元组 …… 39
5.3　字符串 …… 39
5.3.1　各种内置方法 …… 40
5.3.2　格式化 …… 43
5.4　序列 …… 46

**第 6 章　函数** …… 50

6.1　Python 的乐高积木 …… 50
6.1.1　创建和调用函数 …… 50
6.1.2　函数的参数 …… 51
6.1.3　函数的返回值 …… 52
6.2　灵活即强大 …… 52
6.2.1　形参和实参 …… 52

6.2.2 函数文档 …… 52
6.2.3 关键字参数 …… 53
6.2.4 默认参数 …… 53
6.2.5 收集参数 …… 54
6.3 我的地盘听我的 …… 55
6.3.1 函数和过程 …… 55
6.3.2 再谈谈返回值 …… 56
6.3.3 函数变量的作用域 …… 56
6.4 内嵌函数和闭包 …… 58
6.4.1 global 关键字 …… 58
6.4.2 内嵌函数 …… 59
6.4.3 闭包(closure) …… 60
6.5 lambda 表达式 …… 62
6.6 递归 …… 64
6.6.1 递归是"神马" …… 64
6.6.2 写一个求阶乘的函数 …… 66
6.6.3 这帮小兔崽子 …… 68
6.6.4 汉诺塔 …… 70

**第 7 章 字典和集合** …… 72

7.1 字典:当索引不好用时 …… 72
7.1.1 创建和访问字典 …… 72
7.1.2 各种内置方法 …… 74
7.2 集合:在我的世界里,你就是唯一 …… 77
7.2.1 创建集合 …… 78
7.2.2 访问集合 …… 79
7.2.3 不可变集合 …… 79

**第 8 章 永久存储** …… 80

8.1 文件:因为懂你,所以永恒 …… 80
8.1.1 打开文件 …… 80
8.1.2 文件对象的方法 …… 81
8.1.3 文件的关闭 …… 81
8.1.4 文件的读取和定位 …… 82
8.1.5 文件的写入 …… 83
8.1.6 一个任务 …… 83
8.2 文件系统:介绍一个高大上的东西 …… 85
8.3 pickle:腌制一缸美味的泡菜 …… 91

**第 9 章　异常处理** …… 93

9.1　你不可能总是对的 …… 93
9.2　try-except 语句 …… 95
9.2.1　针对不同异常设置多个 except …… 96
9.2.2　对多个异常统一处理 …… 97
9.2.3　捕获所有异常 …… 97
9.3　try-finally 语句 …… 97
9.4　raise 语句 …… 98
9.5　丰富的 else 语句 …… 98
9.6　简洁的 with 语句 …… 99

**第 10 章　图形用户界面入门** …… 101

10.1　导入 EasyGui …… 102
10.2　使用 EasyGui …… 102
10.3　修改默认设置 …… 104

**第 11 章　类和对象** …… 105

11.1　给大家介绍对象 …… 105
11.2　对象＝属性＋方法 …… 105
11.3　面向对象编程 …… 106
11.3.1　self 是什么 …… 107
11.3.2　你听说过 Python 的魔法方法吗 …… 107
11.3.3　公有和私有 …… 108
11.4　继承 …… 109
11.4.1　调用未绑定的父类方法 …… 111
11.4.2　使用 super 函数 …… 112
11.5　多重继承 …… 112
11.6　组合 …… 113
11.7　类、类对象和实例对象 …… 114
11.8　到底什么是绑定 …… 115
11.9　一些相关的 BIF …… 116

**第 12 章　魔法方法** …… 120

12.1　构造和析构 …… 120
12.1.1　__init__(self[, ...]) …… 120
12.1.2　__new__(cls[, ...]) …… 121
12.1.3　__del__(self) …… 122

12.2　算术运算 …… 122
12.2.1　算术操作符 …… 123
12.2.2　反运算 …… 125
12.2.3　增量赋值运算 …… 126
12.2.4　一元操作符 …… 126
12.3　简单定制 …… 126
12.4　属性访问 …… 131
12.5　描述符(property 的原理) …… 135
12.6　定制序列 …… 137
12.7　迭代器 …… 139
12.8　生成器(乱入) …… 142

**第 13 章　模块** …… 145

13.1　模块就是程序 …… 145
13.2　命名空间 …… 146
13.3　导入模块 …… 146
13.4　__name__='__main__' …… 147
13.5　搜索路径 …… 149
13.6　包 …… 150
13.7　像个极客一样去思考 …… 150

**第 14 章　论一只爬虫的自我修养** …… 157

14.1　入门 …… 157
14.2　实战 …… 159
14.2.1　下载一只猫 …… 159
14.2.2　翻译文本 …… 161
14.3　隐藏 …… 166
14.3.1　修改 User-Agent …… 166
14.3.2　延迟提交数据 …… 168
14.3.3　使用代理 …… 169
14.4　Beautiful Soup …… 171
14.5　正则表达式 …… 174
14.5.1　re 模块 …… 175
14.5.2　通配符 …… 175
14.5.3　反斜杠 …… 175
14.5.4　字符类 …… 176
14.5.5　重复匹配 …… 177
14.5.6　特殊符号及用法 …… 178

14.5.7 元字符 …… 180
14.5.8 贪婪和非贪婪 …… 182
14.5.9 反斜杠＋普通字母＝特殊含义 …… 183
14.5.10 编译正则表达式 …… 184
14.5.11 编译标志 …… 184
14.5.12 实用的方法 …… 185
14.6 异常处理 …… 190
14.6.1 URLError …… 190
14.6.2 HTTPError …… 191
14.6.3 处理异常 …… 193
14.7 安装 Scrapy …… 194
14.8 Scrapy 爬虫之初窥门径 …… 196
14.8.1 Scrapy 框架 …… 196
14.8.2 创建一个 Scrapy 项目 …… 198
14.8.3 定义 Item 容器 …… 198
14.8.4 编写爬虫 …… 199
14.8.5 爬 …… 199
14.8.6 取 …… 201
14.8.7 在 Shell 中尝试 Selector 选择器 …… 201
14.8.8 使用 XPath …… 203
14.8.9 提取数据 …… 203
14.8.10 使用 item …… 206
14.8.11 存储内容 …… 207

**第 15 章 GUI 的最终选择：Tkinter** …… 208

15.1 Tkinter 之初体验 …… 208
15.2 Label 组件 …… 210
15.3 Button 组件 …… 212
15.4 Checkbutton 组件 …… 213
15.5 Radiobutton 组件 …… 214
15.6 LabelFrame 组件 …… 215
15.7 Entry 组件 …… 216
15.8 Listbox 组件 …… 221
15.9 Scrollbar 组件 …… 223
15.10 Scale 组件 …… 224
15.11 Text 组件 …… 225
15.11.1 Indexes 用法 …… 227
15.11.2 Marks 用法 …… 229

15.11.3 Tags 用法 …… 231
15.12 Canvas 组件 …… 237
15.13 Menu 组件 …… 242
15.14 Menubutton 组件 …… 245
15.15 OptionMenu 组件 …… 246
15.16 Message 组件 …… 247
15.17 Spinbox 组件 …… 248
15.18 PanedWindow 组件 …… 248
15.19 Toplevel 组件 …… 250
15.20 事件绑定 …… 252
15.21 事件序列 …… 254
15.21.1 type …… 254
15.21.2 modifier …… 255
15.22 Event 对象 …… 256
15.23 布局管理器 …… 258
15.23.1 pack …… 258
15.23.2 grid …… 259
15.23.3 place …… 261
15.24 标准对话框 …… 262
15.24.1 messagebox(消息对话框) …… 262
15.24.2 filedialog(文件对话框) …… 265
15.24.3 colorchooser(颜色选择对话框) …… 266

**第 16 章 Pygame：游戏开发 …… 268**

16.1 安装 Pygame …… 268
16.2 初步尝试 …… 269
16.3 解惑 …… 271
16.3.1 什么是 Surface 对象 …… 271
16.3.2 将一个图像绘制到另一个图像上是怎么一回事 …… 272
16.3.3 移动图像是怎么一回事 …… 273
16.3.4 如何控制游戏的速度 …… 273
16.3.5 Pygame 的效率高不高 …… 273
16.3.6 我应该从哪里获得帮助 …… 274
16.4 事件 …… 274
16.5 提高游戏的颜值 …… 277
16.5.1 显示模式 …… 277
16.5.2 全屏才是王道 …… 278
16.5.3 使窗口尺寸可变 …… 279

16.5.4 图像的变换 …… 279
16.5.5 裁剪图像 …… 281
16.5.6 转换图片 …… 287
16.5.7 透明度分析 …… 287
16.6 绘制基本图形 …… 291
16.6.1 绘制矩形 …… 292
16.6.2 绘制多边形 …… 293
16.6.3 绘制圆形 …… 293
16.6.4 绘制椭圆形 …… 294
16.6.5 绘制弧线 …… 294
16.6.6 绘制线段 …… 295
16.7 动画精灵 …… 296
16.7.1 创建精灵 …… 298
16.7.2 移动精灵 …… 299
16.8 碰撞检测 …… 301
16.8.1 尝试自己写碰撞检测函数 …… 301
16.8.2 sprite 模块提供的碰撞检测函数 …… 303
16.8.3 实现完美碰撞检测 …… 304
16.9 播放声音和音效 …… 305
16.10 响应鼠标 …… 308
16.10.1 设置鼠标的位置 …… 308
16.10.2 自定义鼠标光标 …… 309
16.10.3 让小球响应光标的移动频率 …… 311
16.11 响应键盘 …… 313
16.12 结束游戏 …… 314
16.12.1 发生碰撞后获得随机速度 …… 314
16.12.2 减少“抖动”现象的发生 …… 315
16.12.3 游戏胜利 …… 317
16.12.4 更好地结束游戏 …… 318
16.13 经典飞机大战 …… 319
16.13.1 游戏设定 …… 319
16.13.2 主模块 …… 320
16.13.3 我方飞机 …… 322
16.13.4 响应键盘 …… 322
16.13.5 飞行效果 …… 323
16.13.6 敌方飞机 …… 324
16.13.7 提升敌机速度 …… 325
16.13.8 碰撞检测 …… 326

16.13.9　完美碰撞检测 …… 329
16.13.10　一个 BUG …… 330
16.13.11　发射子弹 …… 331
16.13.12　设置敌机“血槽” …… 333
16.13.13　中弹效果 …… 335
16.13.14　绘制得分 …… 336
16.13.15　暂停游戏 …… 337
16.13.16　控制难度 …… 338
16.13.17　全屏炸弹 …… 339
16.13.18　发放补给包 …… 340
16.13.19　超级子弹 …… 343
16.13.20　三次机会 …… 344
16.13.21　结束画面 …… 347

**参考文献** …… 350

# 第1章 就这么愉快地开始吧

## 1.1 获得 Python





我观察到这么一个现象：很多初学的朋友都会在学习论坛上问什么语言才是最好的？他们的目的很明确，就是要找一门“最好”的编程语言，然后持之以恒地学习下去。没错，这种“执子之手，与子偕老”的专一精神是我们现实社会所推崇的。但在编程的世界里，我们并不提倡这样。我们更提倡“存在即合理”，当前热门的编程语言都有其存在的道理，它们都有各自擅长的领域和适用性。因此我们没办法去衡量哪一门语言才是最好的。

Python 的语法是非常精简的，对于一位完美主义者来说，Python 将是他爱不释手的伙伴。Python 社区的目标就是构造完美的 Python 语言！本书将使用 Python3 来进行讲解，而 Python3 不完全兼容 Python2 的语法，这样做无疑会让大多数程序员心生怨愤且喋喋不休，因为他们用 Python2 写的大量代码经过层层调试已经趋近完美，并已部署到服务器或应用上了。Python3 对 Python2 的语法不兼容，意味着他们的这些应用需要进行转换和重新调试……但是，Python 社区仍旧坚持推出全新的 Python3。只有勇敢地割掉与时代发展不相符的瑕疵部分，才能缔造出真正的完美体验！

工欲善其事，必先利其器。我们要成为“大牛”，要用 Python 去拯救世界，要做的第一件事就是要下载一个 Python 的安装程序并成功地将它安装到你的计算机上。

安装 Python 非常容易，你可以在它的官网找到最新的版本并下载（注：本书所需要的程序、例子均附带在本书配套资源中），地址是 http://www.python.org。

如图 1-1 所示，进入 Python 官网后找到 Download 字样，下载最新版本的 Python 即可。

如果是其他操作系统（例如，Mac OS X），在页面下方可以找到对应的下载地址，如图 1-2 所示。

此处演示的是本书截稿前的最新版本 Python 3.4.3（32 位）（注：这里建议大家安装 32 位版，因为本书第 16 章安装 Pygame 时需要 32 位版本的 Python），一般大家下载最新版本即可。安装 Python3 非常简单，打开下载好的安装包，按照默认选项安装即可。

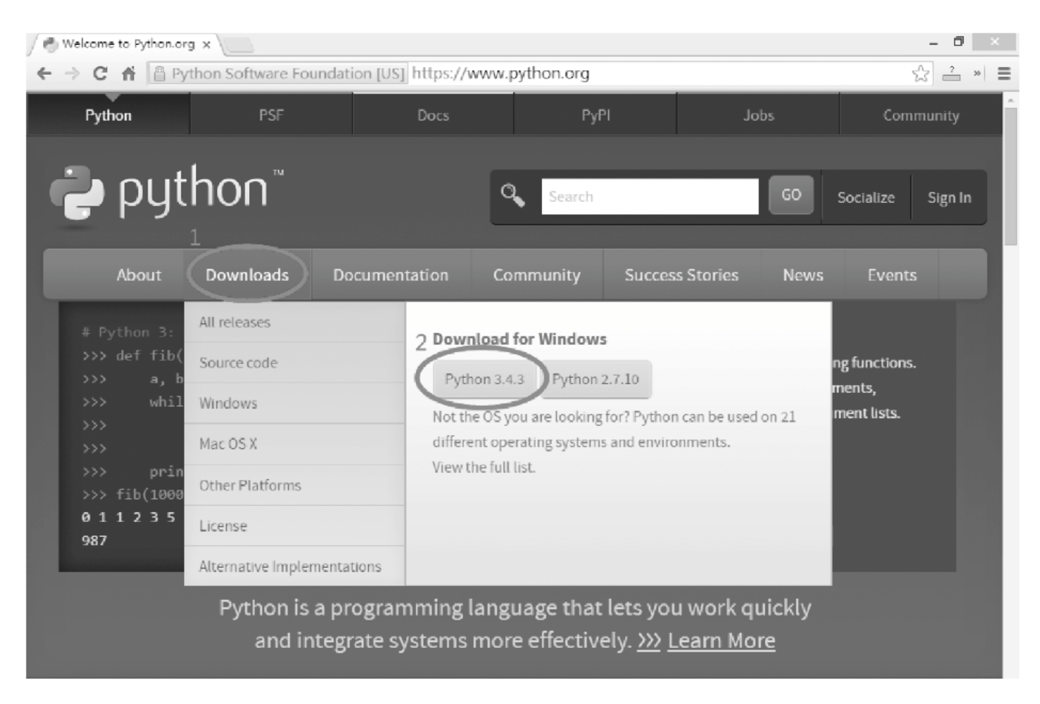



图 1-1　下载 Python3

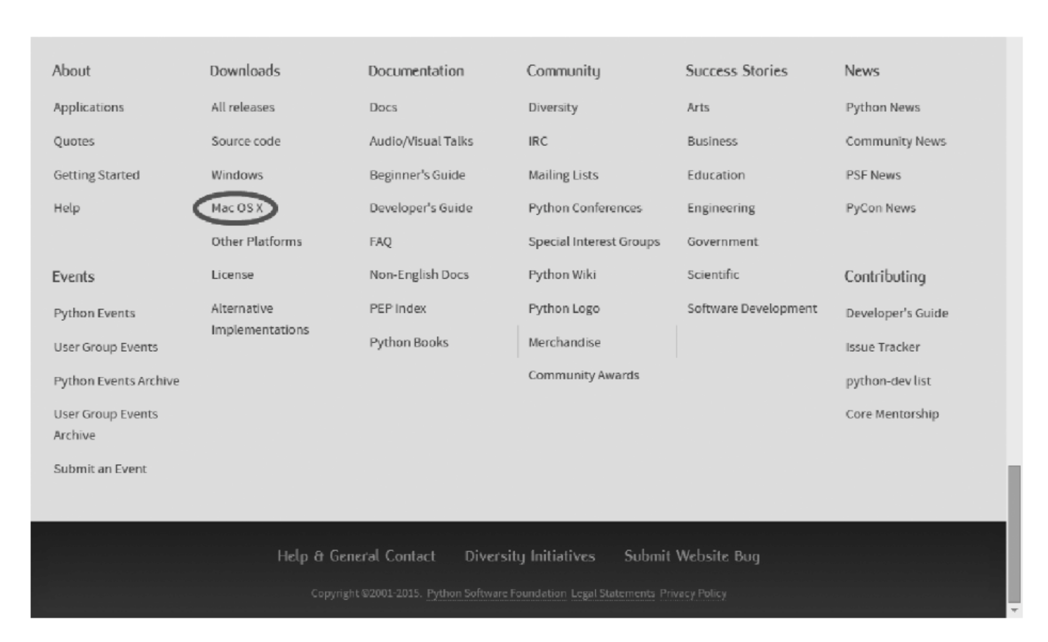



图 1-2　下载 Python3

## 1.2　从 IDLE 启动 Python

IDLE 是一个 Python Shell，shell 的意思就是“外壳”，从基本上说，就是一个通过输入文本与程序交互的途径。像 Windows 的 cmd 窗口，像 Linux 那个黑乎乎的命令窗口，它们都是 shell，利用它们，就可以给操作系统下达命令。同样，可以利用 IDLE 这个 shell 与 Python 进

行互动。

>>>这个提示符含义是：Python 已经准备好了，在等着输入 Python 指令呢。如图 1-3 所示，可以看到 Python 已经按照我们的要求去做了，在屏幕上打印（注：这里打印的意思是“打印”到屏幕上）I love fishc.com 这个充满浓浓爱意的字符串，这说明什么？没错，这说明我们是“爱鱼 C”的，也说明了我们跟 Python 的第一次亲密接触是有感觉的，她完全能够理解我的想法。

```
*Python 3.4.3 Shell*
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("I love FishC.com!")
I love FishC.com!
>>>
```

图 1-3　在 Python 的 IDLE 中输入命令

## 1.3 失败的尝试

像下面这样输入，Python 就会“笨笨地”出错：

```
>>> print "I love fishc.com" # 这是 Python2.x 的语法
SyntaxError: Missing parentheses in call to 'print'
>>> printf("I love fishc.com"); # 这是 C 语言的语法
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    printf("I love fishc.com");
NameError: name 'printf' is not defined
```

其实 Python3 哪里是“笨”，她只是小气，所以显得蠢萌蠢萌的。我们仿佛听到她在说：为什么此时此刻你跟我在一起还想着前任？为什么你跟我在一起还想着其他女人，小 C 她哪点儿比我好，她还要加分号呢，我可不用！

大家看到上边的代码中井号（#）后边加了段中文，井号起到的作用是注释，也就是说，井号后边的内容是给人们看的，并不会被当作代码运行。

## 1.4 尝试点儿新的东西

尝试点儿新的东西，在 IDLE 中输入 print(5+3)或者直接输入 5+3：

```
>>> print(5 + 3)
8
>>> 5 + 3
8
```

看起来 Python 还会做加法！这并不奇怪，因为计算机最开始的时候就是用来计算的，任何编程语言都具备计算能力，那接下来看看 Python 在计算方面有何神奇。

不妨再试试计算 1234567890987654321 * 9876543210123456789：

```
>>> 1234567890987654321 * 9876543210123456789
12193263121170553265523548251112635269
```

怎么样？如果 C 语言实现起来费劲，要九曲十八弯地利用数组做大数运算，在这里 Python 轻而易举就完成了！

还有呢，大家试试输入 print("Well water " + "River")：

```
>>> print("Well water " + "River")
Well water River
```

可以看到，井水和河水又友好地在一起生活了，祝它们幸福吧！

## 1.5 为什么会这样

再试试 print("I love python\n" * 3) ：

```
>>> print("I love python\n" * 3)
I love python
I love python
I love python
```

哇，字符串和数字还可以做乘法，结果是重复显示 N 个字符串。既然乘法可以，那不妨试试加法。print("I love python\n" + 3)：

```
>>> print("I love python\n" + 3)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print("I love python\n" + 3)
TypeError: Can't convert 'int' object to str implicitly
```

失败了！这是为什么呢？大家不妨课后自己思考一下。

# 第2章 用Python设计第一个游戏

## 2.1 第一个小游戏



有读者可能会说："哇，小甲鱼(注：作者)！你开玩笑呢？这么快就教我们开发游戏啦？难道你不打算先讲讲变量、分支、循环、条件、函数等常规的内容？"

没错的，大家如果继续学下去就会发现，本书的教学会围绕着一个个个性鲜明的实例来展开，跟着本书完成这些实例的编写，你会发觉不知不觉中那些该掌握的知识，已经化作你身体的一部分了！这样的学习方式才能充满快乐并让你一直期待下一章节的到来。

好，今天来讲一下"植物大战僵尸"这款游戏的编写……但这是不可能的，因为虽然说Python容易入门，但像"植物大战僵尸"这类游戏要涉及碰撞检测、边缘检查、画面刷新和音效等知识点比较多，需要将这些基础知识累积完成才能开始讲。

目前对于我们所掌握的基础……貌似只有print()这个BIF，哦，BIF的概念甚至还没讲解……不过请淡定，这一点儿也不影响我们今天的节奏！

那么今天是一个什么样的节奏呢？今天打算讲一个文字游戏……

先来看下这段代码，并试图猜测一下每条语句的作用：

```
# p2_1.py
"""--- 第一个小游戏 ---"""
temp = input("不妨猜一下小甲鱼现在心里想的是哪个数字：")
guess = int(temp)
if guess == 8:
    print("你是小甲鱼心里的蛔虫吗?!")
    print("哼,猜中了也没有奖励!")
else:
    print("猜错啦,小甲鱼现在心里想的是8!")
print("游戏结束,不玩啦^_^")
```

在这里要求大家都动动手，亲自输入这些代码，你需要做的是：

- 打开IDLE。
- 选择File->New Window命令(或者你可以直接按Ctrl+N键，在很多地方这个快捷键都是新建一个文件的意思)。
- 按照上边的格式填入代码。
- 按快捷键Ctrl+S，将源代码保存为名为p2_1.py的文件。
- 输完代码一起来体验一下，F5走起(也可以选择Run->Run Module命令)！

程序执行结果如下：

```
>>>
不妨猜一下小甲鱼现在心里想的是哪个数字：5
猜错啦，小甲鱼现在心里想的是 8!
游戏结束，不玩啦^_^
>>>
```

**提示**

> Tab 按键的使用：
> (1) 缩进。
> (2) IDLE 会提供一些建议，例如输入 pr TAB 会显示所有可能的命令供你参考。

OK，我们是看到程序成功跑起来了，但坦白说，这玩意儿配叫游戏吗？呃……没事啦，咱慢慢改进，好，我们说下语法。

有 C-like 语言（一切语法类似 C 语言的编程语言称为 C-like 语言）编程经验的朋友可能会受不了，变量呢？声明呢？怎么直接就给变量定义了呢！有些真正零基础的读者可能还不知道什么是变量，不怕，随着本书内容的展开，大家很快就能掌握相关的知识。有些读者可能发现这个小程序没有任何大括号，好多编程语言都用大括号来表示循环、条件等的作用域，而在 Python 这里是没有的。在 Python 中，只需要用适当缩进来表示即可。

## 2.2 缩进

缩进是 Python 的灵魂，缩进的严格要求使得 Python 的代码显得非常精简并且有层次。但是，在 Python 里对待代码的缩进要十分小心，因为如果没有正确地使用缩进，代码所做的事情可能和你的期望相差甚远（就像在 C 语言里括号打错了位置）。

如果在正确的位置输入冒号（:），IDLE 会在下一行自动进行缩进。正如方才的代码，在 if 和 else 语句后边加上冒号（:），然后按下回车，第二行开始的代码会自动进行缩进。if 条件下边有两个语句分别有缩进，那么说明这两个语句是属于 if 条件成立后所需要执行的语句，换句话说，如果 if 条件不成立，那么两个缩进的语句就不会被执行。

**提示**

> if…else…是一个条件分支，if 后边跟的是条件，如果条件成立，就执行以下缩进的所有内容；如果条件不成立，有 else 的话就执行 else 下缩进的所有内容。条件分支的内容在后边还会做详细的介绍。

## 2.3 BIF

接下来学习一个新的名词：BIF。

BIF 就是 Built-in Functions，内置函数的意思。什么是内置函数呢？为了方便程序员快

速编写脚本程序(脚本就是要代码编写速度快快快!),Python 提供了非常丰富的内置函数,只需要直接调用即可,例如 print()是一个内置函数,它的功能是"打印到屏幕",就是说把括号里的内容显示到屏幕上。input()也是一个 BIF,它的作用是接收用户输入并将其返回,在上方的代码中,用 temp 这个变量来接收。Python 的变量是不需要事先声明的,直接给一个合法的名字赋值,这个变量就生成了。

提示

在 IDLE 中输入 dir(__builtins__)可以看到 Python 提供的内置函数列表。

help()这个 BIF 用于显示 BIF 的功能描述:

```
>>> help(print)
Help on built - in function print in module builtins:

print(...)
    print(value, ..., sep = ' ', end = '\n', file = sys.stdout, flush = False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file - like object (stream); defaults to the current sys.stdout.
    sep: string inserted between values, default a space.
    end: string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

有些读者可能会说,太多 BIF 学不过来记不过来怎么办?看不懂英文说明怎么办?Python3 的资料太少怎么办?大家不用担心,在接下来的每一个环节,作者都会教大家几个常用的 BIF 的用法,然后在课后作业(注:每节课对应的课后作业需要在鱼 C 论坛完成:http://bbs.fishc.com/forum-243-1.html)中强化大家的记忆。所以,大家只要严格跟着作者的脚步走,课后练习坚持自己独立完成,相信即使觉得自己记性很差的朋友,也可以做到倒背如流!

# 第3章 成为高手前必须知道的一些基础知识

## 3.1 变量



在改进小游戏之前，有些必须掌握的知识需要来讲解一下。

当你把一个值赋值给一个名字时，它会存储在内存中，把这块内存称为变量(variable)。在大多数语言中，都把这种行为称为“给变量赋值”或“把值存储在变量中”。

不过，Python与大多数其他计算机语言的做法稍有不同，它并不是把值存储在变量中，而更像是把名字“贴”在值的上边。所以有些Python程序员会说Python没有变量，只有名字。变量就是一个名字，通过这个名字，可以找到我们想到的东西。

看个例子：

```
>>> teacher = "小甲鱼"
>>> print(teacher)
小甲鱼
>>> teacher = "老甲鱼"
>>> print(teacher)
老甲鱼
```

变量为什么不叫“恒量”而叫变量？正是因为它是可变的！再看另一个例子：

```
>>> x = 3
>>> x = 5
>>> y = 8
>>> z = x + y
>>> print(z)
13
```

上面的例子先创建一个变量，名字叫x，给它初始化赋值为3，然后又给它赋值为5(此时3就被5替换掉)，接下来创建另外一个变量y，并初始化赋值为8，最后创建第三个变量z，它的值是变量x和y的和。

同样的方式也可以运用到字符串中：

```
>>> myteacher = "小甲鱼"
>>> yourteacher = "老甲鱼"
>>> ourteacher = myteacher + yourteacher
```

```
>>> print(ourteacher)
小甲鱼老甲鱼
```

这种字符串加字符串的语法，在 Python 里称为字符串的拼接。

需要注意的地方：

- 在使用变量之前，需要对其先赋值。
- 变量名可以包括字母、数字、下划线，但变量名不能以数字开头，这跟大多数高级语言一样——受 C 语言影响，或者说 Python 这门语言本身就是由 C 语言写出来的。
- 字母可以是大写或小写，但大小写是不同的。也就是说，fishc 和 FishC 对于 Python 来说是完全不同的两个名字。
- 等号(=)是赋值的意思，左边是名字，右边是值，不可写反了。
- 变量的命名理论上可以取任何合法的名字，但作为一个优秀的程序员，请尽量给变量取一个专业一点儿的名字。

## 3.2 字符串

到目前为止，我们所认知的字符串就是引号内的一切东西，我们也把字符串叫作文本，文本和数字是截然不同的。

如果直接让两个数字相加，那么 Python 会直接将数字相加后的结果告诉你：

```
>>> 5 + 8
13
```

但是如果在数字的两边是加上了引号，就变成了字符串的拼接，这正是引号带来的差别：

```
>>> '5' + '8'
'58'
```

要告诉 Python 你在创建一个字符串，就要在字符两边加上引号，可以是单引号或者双引号，Python 表示在这一点上不挑剔。但必须成对，你不能一边用单引号，另一边却花心地用上双引号结尾，这样 Python 就不知道你到底想干嘛了：

```
>>> 'Python I love you!"
SyntaxError: EOL while scanning string literal
```

这就有点像你一边跟 Python 说我爱你，一边却搂着小 C，所以，面对这么完美的语言，我们不写别扭的语法！

那如果字符串内容中需要出现单引号或双引号怎么办？

```
>>> 'Let's go'
SyntaxError: invalid syntax
```

像上边这样写 Python 会误会你的意思，从而产生错误。

有两种方法。第一种比较常用，就是使用转义符号(\)对字符串中的引号进行转义：

```
>>> 'Let\'s go'
"Let's go"
```

还有一种方法就是利用 Python 既可以用单引号也可以用双引号表示字符串这一特点，只要用上不同的引号表示字符串，那么 Python 就不会误解你的意思啦。

```
>>> "Let's go"
"Let's go"
```

## 3.3 原始字符串

听起来好像反斜杠是一个好东西，但不妨试试打印 C:\now：

```
>>> string = 'C:\now'
>>> string
'C:\now'
>>> print(string)
C:
ow
```

打印结果并不是我们预期的，原因是反斜杠(\)和后边的字符(n)恰好转义之后构成了换行符(\n)。这时候有朋友可能会说："用反斜杠来转义反斜杠不就可以啦?"嗯，不错，可以用反斜杠对自身进行转义：

```
>>> string = 'C:\\now'
>>> string
'C:\\now'
>>> print(string)
C:\now
```

但如果对于一个字符串中有很多个反斜杠，我们就不乐意了。毕竟，这不仅是一个苦差事，还可能使代码变得混乱。

不过大家也不用怕，因为在 Python 中有一个快捷的方法，就是使用原始字符串。原始字符串的使用非常简单，只需要在字符串前边加一个英文字母 r 即可：

```
>>> string = r'C:\now'
>>> string
'C:\\now'
>>> print(string)
C:\now
```

在使用字符串时需要注意的一点是：无论是否原始字符串，都不能以反斜杠作为结尾(注：反斜杠放在字符串的末尾表示该字符串还没有结束，换行继续的意思，下一节会讲这个内容)。如果你坚持这样做就会报错：

```
>>> string = 'FishC\'
SyntaxError: EOL while scanning string literal
>>> string = r'FishC\'
SyntaxError: EOL while scanning string literal
```

大家不妨考虑一下：如果非要在字符串的结尾加个反斜杠，有什么办法可以灵活实现吗？

## 3.4 长字符串

如果希望得到一个跨越多行的字符串，例如：

从明天起，做一个幸福的人
喂马，劈柴，周游世界
从明天起，关心粮食和蔬菜
我有一所房子，面朝大海，春暖花开

从明天起，和每一个亲人通信
告诉他们我的幸福
那幸福的闪电告诉我的
我将告诉每一个人

给每一条河每一座山取一个温暖的名字
陌生人，我也为你祝福
愿你有一个灿烂的前程
愿你有情人终成眷属
愿你在尘世获得幸福
我只愿面朝大海，春暖花开

嗯，看得出这是一首非常有文采的诗，那如果要把这首诗打印出来，用学过的知识，就不得不使用多个换行符：

```
>>> print("从明天起，做一个幸福的人\n喂马，劈柴，周游世界\n从明天起，关心粮食和蔬菜\n我有一所房子，面朝大海，春暖花开\n\n从明天起，和每一个亲人通信\n告诉他们我的幸福\n那幸福的闪电告诉我的\n我将告诉每一个人\n\n给每一条河每一座山取一个温暖的名字\n陌生人，我也为你祝福\n愿你有一个灿烂的前程\n愿你有情人终成眷属\n愿你在尘世获得幸福\n我只愿面朝大海，春暖花开\n")
从明天起，做一个幸福的人
喂马，劈柴，周游世界
从明天起，关心粮食和蔬菜
我有一所房子，面朝大海，春暖花开
… # 由于篇幅有限,这里省略打印的内容
```

如果行数非常多，又会给我们带来不小的困扰了……好在 Python 总是设身处地地为我们着想——只需要使用三重引号字符串("""内容""")就可以轻松解决问题：

```
>>> print("""
从明天起，做一个幸福的人
喂马，劈柴，周游世界
从明天起，关心粮食和蔬菜
我有一所房子，面朝大海，春暖花开

从明天起，和每一个亲人通信
告诉他们我的幸福
```

```
那幸福的闪电告诉我的
我将告诉每一个人

给每一条河每一座山取一个温暖的名字
陌生人，我也为你祝福
愿你有一个灿烂的前程
愿你有情人终成眷属
愿你在尘世获得幸福
我只愿面朝大海，春暖花开
""")

从明天起，做一个幸福的人
喂马，劈柴，周游世界
从明天起，关心粮食和蔬菜
我有一所房子，面朝大海，春暖花开
… # 篇幅有限,这里省略打印的内容
```

最后需要提醒大家的是,编程的时候,时刻要注意 Speak English! 初学者最容易犯的错误(没有之一)就是误用了中文的标点符号。切记:编程中使用的标点符号都是英文的!

## 3.5 改进我们的小游戏



不得不承认,之前的小游戏真的是太简单了。有很多朋友为此提出了不少的建议,小甲鱼做了一下总结,大概有以下几个方面需要改进:

(1) 当用户猜错的时候程序应该给点提示,比如告诉用户当然输入的值比答案是大了还是小了。

(2) 每运行一次程序只能猜一次,应该提供多次机会给用户猜测,至少要三次嘛,人非圣贤,孰能一击即中,你说是吧?!

(3) 每次运行程序,答案可以是随机的。因为程序答案固定,容易导致答案外泄,比如小红玩了之后知道正确答案是 8,就可能会把结果告诉小明,小明又会乱说。所以希望游戏的答案可以是随机的。

这些挑战对于如此聪明的读者来说一定不成问题,让我们抄起家伙(Python)一个个来解决掉!

## 3.6 条件分支

第一个改进要求:当用户猜错的时候程序应该给点提示,比如告诉用户当然输入的值比答案是大了还是小了。程序改进后(假如答案是 8):

- 如果用户输入 3,程序应该提示比答案小了。
- 如果用户输入 9,程序应该提示比答案大了。

这就涉及一个比较的问题了,作为初学者,可能不大熟悉计算机是如何进行比较吧?但我想大家都一定认识大于号(>)、小于号(<)以及等于号(==)(注:在 Python 中,用两个连续等号表示等于号,用单独一个等号表示赋值,还记得吧?那不等于呢?嗯,不等于这个有点特

殊,用感叹号和一个等号搭配来表示)。

另外,还需要掌握 Python 的比较操作符有:

```
<,<=,>,>=,==,!=
```

在 IDLE 中输入两个数以及比较操作符,Python 会返回比较后的结果:

```
>>> 1 < 3
True
>>> 1 > 3
False
>>> 1 == 3
False
>>> 1 != 3
True
```

这里 1 和 3 进行比较,判断 1 是否小于 3,在小于号左右分别留了一个空格,这不是必需的,但代码量一多,看上去会美观很多。Python 是一个注重审美的编程语言,这就跟人一样,人长得怎样是天生的,一般无法改变,但人的气质修养是可以从每个细小动作看出来的,人们说心灵美才是美,指的就是这一方面。程序也一样,你可以不修边幅、邋邋遢遢,只求不出错误,但别人阅读你的代码时很难受,他就不愿跟你一起合作开发,要是你的代码工整,注释得当,远远看上去犹如大家之作,那结果肯定不用说啦!

大家还记得 if-else 吧?如果程序仅仅只是一个命令清单的话,那么他只需要笔直地一条路走到黑,但至少觉得应该把程序设计得更聪明点——可以根据不同的条件执行不同的任务,这就是条件分支。

```
if 条件 :
    条件为真(True)执行的操作
else:
    条件为假(False)执行的操作
```

那现在让我们把第 1 个要求的代码写出来吧:

```
if guess == secret:
    print("哎呀,你是小甲鱼肚里的蛔虫吗?!")
    print("哼~猜中了也没有奖励!")
else:
    if guess > secret:
        print("哥,大了大了~~~")
    else:
        print("嘿,小了小了~~~")
```

## 3.7 while 循环

第 1 个要求实现了,可是用户还不高兴,他们会抱怨道:“为什么我要不停地重新运行你这个程序呢?难道你不能每次运行多给几次输入的机会吗?”(我们这个程序还好,几次尝试就可以成功了,但如果范围扩大为 1~100,那么尝试的次数就要随之增加,总让用户不断地重新打开程序,这种程序的体验未免就太差了哈!)

第 2 个改进要求：程序应该提供多次机会给用户猜测，专业点来讲就是程序需要重复运行某些代码。

下面介绍 Python 的 while 循环语法。

```
while 条件:
    条件为真(True)执行的操作
```

非常简单，对吧？Python 一向就是这么简单，那一起来修改代码吧：

```
while guess != 8:
    temp = input("哎呀,猜错了,请重新输入吧: ")
    guess = int(temp)

    if guess == 8:
        print("哎呀,你是小甲鱼肚里的蛔虫吗?!")
        print("哼～猜中了也没有奖励!")
    else:
        if guess > 8:
            print("哥,大了大了～～～")
        else:
            print("嘿,小了小了～～～")
```

聪明的读者可能已经发现了，这么改的话，程序的意思是只有用户输入正确的数字循环才能结束。这好像跟我们的第 2 个要求有点不同了，所以大家不妨边思考边动手，看怎么改才是正确的。

给大家一点提示，大家思考一下如何修改，这里我给大家的提示是：使用 and 逻辑操作符。Python 的逻辑操作符可以将任意表达式连接在一起，并得到一个布尔类型的值。布尔类型只有两个值：True 和 False，就是真与假，来看下面的例子：

```
>>> (3 > 2)and(1 < 2)
True
>>> (3 > 2)and(1 > 2)
False
```

很明显，1>2 这个条件是个伪命题，所以 and 的结果为假。用 and 逻辑操作符运行，只有当两边的条件均为真时，结果才能是 True，否则为 False，大家可以自己多做几次实验来证明。

## 3.8 引入外援

第 3 个改进要求：每次运行程序，答案是随机的。需要怎么实现呢？需要引入外援：random 模块，

等等，模块这个名字怎么那么熟悉？

啊哈！想起来了，每次写完程序的时候，都要按一下快捷键 F5 运行，那里就显示着 RUN MODULE，我们编写的程序实际上就是一个模块，只是我们目前还没有发觉。

那这个 random module 里边有一个函数叫作 randint()，它会返回一个随机的整数。可以利用这个函数来改造我们的游戏：

```
# p3_1.py
```

```
import random

secret = random.randint(1,10)
temp = input("不妨猜一下小甲鱼现在心里想的是哪个数字：")
guess = int(temp)

while guess != secret:
    temp = input("哎呀,猜错了,请重新输入吧：")
    guess = int(temp)

    if guess > secret:
        print("哥,大了大了～～～")
    else:
        print("嘿,小了小了～～～")

    if guess == secret:
        print("哎呀,你是小甲鱼心里的蛔虫吗?!")
        print("哼～猜中了也没有奖励!")

print("游戏结束,不玩啦^_^")
```

## 3.9 闲聊数据类型



所谓闲聊，也称为 gossip，就是一点小事可以聊上半天。下面就来聊一聊 Python 的数据类型。

在此之前，你可能已经听说过，咱这个 Python 的变量是没有类型的。对，没错，小甲鱼也曾经说过，Python 的变量更像是名字标签，想贴哪儿就贴哪儿。通过这个标签，就可以轻易找到变量在内存中对应的存放位置了。

但这绝不是说 Python 就没有数据类型这回事。大家还记得'520'和 520 的区别吗？

没错，带了引号的，无论是双引号还是单引号或者是三引号，都是字符串；而不带引号的，就是数字。字符串相加叫作拼接，咳咳，不是拼爹，是拼接！数字相加就会得到两个数字的和：

```
>>> '520' + '1314'
'5201314'
>>> 520 + 1314
1834
```

Python 有很多重要的数据类型，不过这里不会一下子全都扔给大家。因为一来你肯定一下子记不了那么多；另外现在所要掌握的知识还不需要这么多的数据类型来配合实现。所以，每个阶段所学习的内容都是必要的，我们也只学习那些必要的内容。

Python 的字符串类型已经简单讲过，后边还会对字符串进行深入的探讨，所以大家别吐槽小甲鱼怎么都是浅尝辄止，没有那回事儿！咱只是分阶段逐步渗透，逐层进行消化，一下子说太深入，大家消化不了，教学也会变成纯理论化(小甲鱼知道死板的模式是大家最讨厌的)。

今天来介绍一些 Python 的数值类型又包含整型、浮点型、布尔类型、复数类型等。

### 3.9.1 整型

整型说白了就是平时所见的整数，Python3 的整型已经与长整型进行了无缝结合，现在的

Python3 的整型类似于 Java 的 BigInteger 类型，它的长度不受限制，如果说非要有个限制，那只限于计算机的虚拟内存总数。所以用 Python3 很容易进行大数计算。

## 3.9.2 浮点型

浮点型就是平时所说的小数，例如圆周率 3.14 是浮点型，例如地球到太阳的距离大约 1.5 亿千米，也是浮点型。Python 区分整型和浮点型的唯一方式，就是看有没有小数点。

谈到浮点型，不得不说下 E 记法。E 记法也就是平时所说的科学计数法，用于表示特别大和特别小的数：

```
>>> a = 0.0000000000000000000025
>>> a
2.5e-21
```

对于地球到太阳的距离 1.5 亿千米，如果转换成米的话，那就是一个非常大的数了(150 000 000 000)，但是如果你用 E 记法就是 1.5e11(大 E 和小 e 都可以)。

其实大家应该已经发现了，这个 E 的意思是指数，指底数为 10，E 后边的数字就是 10 的多少次幂。像 15 000 等于 1.5×10 000，也就是 $1.5\times10^4$，E 记法写成 1.5e4。

## 3.9.3 布尔类型

布尔类型事实上是特殊的整型，尽管布尔类型用 True 和 False 来表示“真”与“假”，但布尔类型可以当作整数来对待，True 相当于整型值 1，False 相当于整型值 0，因此下边这些运算都是可以的(最后的例子报错是因为 False 相当于 0，而 0 不能作为除数)。

```
>>> True + True
2
>>> True * False
0
>>> True / False
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    True / False
ZeroDivisionError: division by zero
```

当然把布尔类型当成 1 和 0 来参与运算这种做法是不妥的，这跟你把羊驼当成是一种马一样，所以大家知道就好，千万别在实际应用中这么去做！

## 3.9.4 类型转换

接下来介绍几个跟数据类型紧密相关的函数：int()、float()和 str()。

int()的作用是将一个字符串或浮点数转换为一个整数：

```
>>> a = '520'
>>> b = int(a)
>>> a, b
('520', 520)
>>> c = 5.99
>>> d = int(c)
```

```
>>> c, d
(5.99, 5)
```

注意了,如果是浮点数转换为整数,那么 Python 会采取“截断”处理,就是把小数点后的数据直接砍掉,注意不是四舍五入哦!

float()的作用是将一个字符串或整数转换成一个浮点数(就是小数啦):

```
>>> a = '520'
>>> b = float(a)
>>> a, b
('520', 520.0)
>>> c = 520
>>> d = float(c)
>>> c, d
(520, 520.0)
```

str()的作用是将一个数或任何其他类型转换成一个字符串:

```
>>> a = 5.99
>>> b = str(a)
>>> b
'5.99'
>>> c = str(5e15)
>>> c
'5000000000000000.0'
```

### 3.9.5　获得关于类型的信息

有时候可能需要确定一个变量的数据类型,例如用户的输入,当需要用户输入一个整数,但用户却输入一个字符串,就有可能引发一些意想不到的错误或导致程序崩溃!

现在告诉大家一个好消息,Python 其实提供了一个函数,可以明确告诉我们变量的类型,这就是 type()函数:

```
>>> type('520')
<class 'str'>
>>> type(5.20)
<class 'float'>
>>> type(5e20)
<class 'float'>
>>> type(520)
<class 'int'>
>>> type(True)
<class 'bool'>
```

当然,通向罗马的道路非常多,无须在一棵树上吊死,查看 Python 的帮助文档,它更建议我们使用 isinstance()这个 BIF 来确定变量的类型。这个 BIF 有两个参数:第一个是待确定类型的数据;第二个是指定一个数据类型。

isinstance()会根据两个参数返回一个布尔类型的值,True 表示类型一致,False 表示类型不一致:

```
>>> a = "小甲鱼"
>>> isinstance(a, str)
True
>>> isinstance(520, float)
False
>>> isinstance(520, int)
True
```

## 3.10 常用操作符



### 3.10.1 算术操作符

和绝大多数编程语言一样，Python 的算术操作符大部分和我们理解的一样，注意，这里说的是大部分，不是全部：

```
+  -  *  /  %  **  //
```

前边四个就不用介绍啦，加减乘除，大家都懂。不过有点小技巧倒不是大家都知道。

例如，当你想对一个变量本身进行算术运算的时候，你是不是会觉得写 a = a + 1 或 b = b - 3 这类操作符特别麻烦？没错，在 Python 中可以做一些简化：

```
>>> a = b = c = d = 10
>>> a += 1
>>> b -= 3
>>> c * = 10
>>> d / = 8
>>> print(a, b, c, d)
11 7 100 1.25
```

如果使用过 Python2. x 版本的读者可能会发现，咱 Python 的除法变得有些不同了。包括很多编程语言，整数除法一般都是采用 floor 的方式，有些书籍称为地板除法（注：因为 floor 的翻译就是地板的意思）。地板除法的概念是：计算结果取比商小的最大整型，也就是舍弃小数的意思（注：例如 3 / 2 等于 1）。但是在这里我们发现，即使是进行整数间的除法，但是答案是自动返回一个浮点型的精确数值，也就是 Python 用真正的除法代替了地板除法。

那有些朋友不乐意了，他说"萝卜青菜各有所爱，我就喜欢原来的除法，我觉着整数除以整数就应该得到一个整数嘛。"Python 团队也为此想好了后路，就是大家看到的双斜杠，它执行的就是地板除法的操作，不过要注意一点的是，无论是整数运算还是浮点数运算，都会执行地板除法：

```
>>> 3 // 2
1
>>> 3.0 // 2
1.0
```

关于 Python3 在除法运算上的改革，支持的和谩骂的几乎各占一半，有些人支持这种做法，因为 Python 的除法运算从一开始的设计就有失误，但有些人又不想因此修改自己的海量代码，而剩下的人则想要真正的除法。无论怎样，Python 团队是秉承着追求完美和卓越的思

维去一次次改进 Python 这门编程语言，所以小甲鱼说 Python3 已经是非常棒的版本了。

百分号(%)表示求余数的意思：

```
>>> 5 % 2
1
>>> 4 % 2
0
>>> 520 % 14
2
```

### 3.10.2 优先级问题

当一个表达式存在着多个运算符的时候，就可能会发生以下对话。

加法运算符说："我先到的，我先计算！"

乘法运算符说："哥我干一次够你翻几个圈了，哥先来！"

减法运算符说："你糊涂了，我现在被当成负号使用，没有我，你们再努力，结果也是得到相反的数！"

除法运算符这时候默默地说："抢吧抢吧，老娘我除以零，大不了大家同归于尽！"

呃！为了防止以上矛盾的出现，我们规定了运算符的优先级，当多个运算符同时出现在一个表达式的时候，严格按照优先级规定的级别来进行运算。

先乘除，后加减，如有括号先运行括号里边的。没错，从小学我们就学到了运算符优先级的精髓，在编程中也是这么继承下来的。例如：

```
-3 * 2 + 5 / -2 - 4
```

相当于

```
(-3) * 2 + 5 / (-2) - 4
```

其实这个多做练习自然就记住了，不用刻意去背。当然在适当的地方加上括号强调一下优先级小甲鱼觉得会是更好的方案。

Python 还有一个特殊的乘法，就是双星号(**)，也称为幂运算操作符。例如 3 ** 2，双星号左侧的 3 称为底数，右侧的 2 称为指数，把这样的算式叫作 3 的 2 次幂，结果就是 3 * 3==9。

在使用 Python 进行幂运算的时候，需要注意的一点是优先级问题，因为幂运算操作符和一元操作符[①]的优先级关系比较特别：幂运算操作符比其左侧的一元操作符优先级高，比其右侧的一元操作符优先级低：

```
>>> -3 ** 2
-9
>>> 3 ** -2
0.1111111111111111
```

### 3.10.3 比较操作符

比较操作符包括：

① 例如减号被当作表示负数的符号来用的时候，它就是一元操作符，因为它只有一个操作数嘛！

```
<  <=  >  >=  ==  !=
```

这个之前讲过了，比较操作符根据表达式的值的真假返回布尔类型的值。这里不重复讲解，列举出来给大家回顾一下而已。

## 3.10.4 逻辑操作符

逻辑操作符包括：

```
and  or  not
```

and 操作符之前已经学习过，在实例中也多次使用，当只有 and 操作符左边的操作数为真，且右边的操作符同时为真的时候，结果为真。

or 操作符跟 and 操作符不同，or 操作符只需要左边或者右边任意一边为真，结果都为真；只有当两边同时为假，结果才为假。

not 操作符是一个一元操作符，它的作用是得到一个和操作数相反的布尔类型的值：

```
>>> not True
False
>>> not 0
True
>>> not 4
False
```

噢，对了，你可能会看到这样的表达式：

```
3 < 4 < 5
```

这在其他编程语言一般是不合法的，但在 Python 中是行得通的，它事实上被解释为：

```
3 < 4 and 4 < 5
```

最后，将目前接触的所有操作符的优先级合并在一起，如图 3-1 所示。

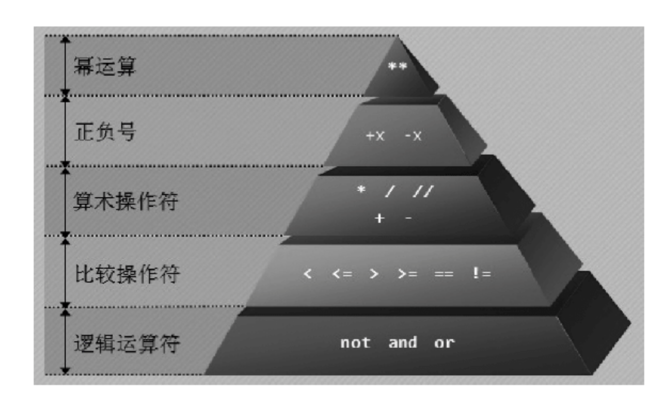



图 3-1　Python 操作符优先级

# 第4章 了不起的分支和循环

## 4.1 分支和循环



有人说，了不起的C语言，因为“机器码生汇编，汇编生C，C生万物”，它几乎铸造了如今IT时代的一切，它是一切的开端，并且仍然没被日新月异的时代所淘汰。

有人可能会反对，因为首先C语言不是世界上第一门编程语言，它仍然要被降级为汇编语言再到机器语言才能为计算机所理解。

这话题扯得有点太远了，小甲鱼想说的是，其实很多初学者会对编程语言有一种莫名其妙的崇拜感？所以呢，他们必须要找出一门全世界公认最牛的语言再来学习好它。

其实，世界上根本没有最优秀的编程语言，只有最合适的语言，面对不同的环境和需求，就会有不同的编程工具去迎合。

今天的主题是“了不起的分支和循环”，为什么小甲鱼不说C语言，不说Python了不起，却毫不吝啬地对分支和循环这两个知识点那么“崇拜”呢？

大家在前面也接触了最简单的分支和循环的使用，那么小甲鱼希望大家思考一下：如果没有分支和循环，我们的程序会变成怎样？

没错，就会变成一堆从上到下依次执行、毫无趣味的代码！还能实现算法吗？当然不能！

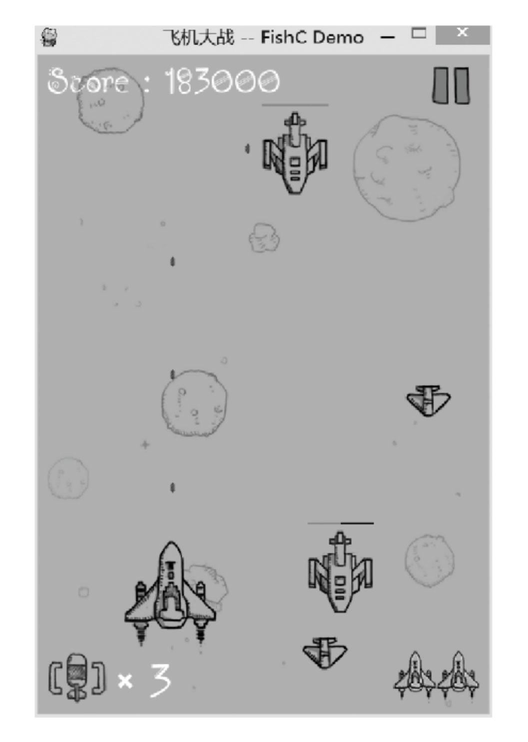
幸好，所有能称得上编程语言的，都应该拥有分支和循环这两种实现。接下来从游戏的角度来谈谈，“打飞机”游戏相信大家非常熟悉了，如图4-1所示。

那么现在就从打飞机来解释一般游戏的组成和架构。

首先进入游戏，很容易发现其实就是进入一个大循环，虽然小甲鱼现在跟大家讨论的是打飞机，但基本上每一个游戏的套路都是一样的，甚至操作系统的消息机制使用的也是同样一个大循环来完成的。游戏中，只要没有触发死亡机制(注：这个游戏的死亡机制是撞到敌机)，敌机都会不断地生成，这足以证明整个游戏就是在一个循环中执行的。

接下来来看一下分支的概念，分支也就是所习惯使用的if条件判断，在条件持续保持成立或不成立的情况下，我们都执行固定的流程。一旦条件发生了改变，原来成立的条件就变成不成立，那么程序就走入另一条路了。就好比如拿我们的飞机去撞击敌机……如图4-2所示。

另外，大家有没有发现，小飞机都是一个样子的？嗯，这说明它们来自于同一个对象的复制品，Python是面向对象的编程，对象这个概念无时无刻不融入在Python的血液里，只是暂时还没有接触这个概念，所以有些朋友还意识不到，不用急，以后会详细讲解这个概念的。



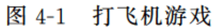


图 4-1　打飞机游戏




图 4-2　打飞机游戏结束界面

最后我要不要告诉大家这个小游戏就只是用了几个循环和 if 条件就写出来啦？没错，编程其实就是这么简单。当然大家要达到自己可以动手写一个界面小游戏的水平，还需要掌握更多的知识！现在需要大家一起来动手，按照刚才看到的小游戏，请拿出纸和笔，将它的实现逻辑尝试给勾画出来(可以使用文字描述，现在只谈框架，不谈代码)。

参考框架如下：

```
加载背景音乐
播放背景音乐
我方飞机诞生
interval = 0

while True:
    if 用户是否单击了关闭按钮:
        退出程序

    interval += 1

    if interval == 50:
        小飞机诞生
        小飞机移动一个位置
        屏幕刷新
        interval = 0

    if 用户鼠标产生移动:
        我方飞机中心位置 = 用户鼠标位置
        屏幕刷新
```

```
    if 我方飞机与小飞机发生肢体接触:
        我方挂,播放撞机音乐
修改我方飞机图案
        打印"GAME OVER"
        停止背景音乐
```

## 4.2 课堂小练习



前面教大家如何正确打飞机,其要点就是:判断和循环,判断就是应不应该做某事,循环就是持续做某事。条件分支,也就是判断,习惯用到的是 if-else 的搭配,而循环就用我们已经掌握了的 while 语句。

现在来考考大家:按照 100 分制,90 分以上成绩为 A,80～90 为 B,60～80 为 C,60 以下为 D。现在要求你写一个程序,当用户输入分数,自动转换为 A、B、C 或 D 的形式打印。

```
# p4_1.py
score = int(input('请输入一个分数:'))
if 100 >= score >= 90:
    print('A')
if 90 > score >= 80:
    print('B')
if 80 > score >= 60:
    print('C')
if 60 > score >= 0:
    print('D')
if score < 0 or score > 100:
    print('输入错误!')
```

当然你也可以写成:

```
# p4_2.py
score = int(input('请输入您的分数:'))
if 100 >= score >= 90:
    print('A')
else:
    if 90 > score >= 80:
        print('B')
    else:
        if 80 > score >= 60:
            print('C')
        else:
            if 60 > score >= 0:
                print('D')
            else:
                print('输入错误!')
```

如果是这样写,条件多了可能会有诸多不便,你完全可以偷懒一下:

```
# p4_3.py
score = int(input('请输入一个分数:'))
if 100 >= score >= 90:
```

```
    print('A')
elif 90 > score >= 80:
    print('B')
elif 80 > score >= 60:
    print('C')
elif 60 > score >= 0:
    print('D')
else:
    print('输入错误!')
```

## 4.3 结果分析

假设输入的分数是98,按照第一种方法是第一次判断成立,接着打印字母A,但接着会进行第二、三、四、五次判断,然后条件都不符合,退出程序。

若采用第二、第三种方法,那么在第一次判断成立并打印字母A后,接着不需要再进行任何判断就可以直接退出程序。可见虽然是很简单的例子,但就输入的98来说,假设每一次判断会消耗一个CPU时间,那么第一种方法比第二和第三种方法多消耗了400%的时间!

所以要实现一个程序事实上并不难,但作为一个优秀的程序员,你必须要形成良好的编程思维。而Python这门语言本身就可以锻炼你这方面的能力。不信?来看下一个问题:Python可以有效避免"悬挂else"。

## 4.4 Python可以有效避免"悬挂else"

什么叫"悬挂else"?举个例子,初学C语言的朋友可能很容易被以下代码欺骗。

```
if ( hi > 2 )
    if( hi > 7 )
        printf("好棒!好棒!");
else
    printf("切~");
```

在这个例子中,虽然else是想和外层的if匹配,但事实上按照C语言的就近匹配原则这个else是属于内层if的。由于初学者的一不小心,就容易导致BUG的出现。这就是著名的"悬挂else"。

而Python的缩进使用强制规定使得代码必须正确对齐,让程序员来决定else到底属于哪一个if。限制你的选择从而减少了不确定性,Python鼓励你第一次就写出正确的代码。所以在Python中制造出"悬挂else"的问题是不可能的。而且,强制使用正确的缩进,Python的代码变得整洁易读,这就是大家都喜欢Python的原因。

## 4.5 条件表达式(三元操作符)

我们说"多少元"操作符的意思是这个操作符有多少个操作数。例如赋值操作符"="是二元操作符,所以它有左边和右边两个操作数。例如"-"是一元操作符,它表示负号,因为只有一

个操作数。那么三元操作符就应该有三个操作数咯？没错的，你猜对了。

其实 Python 的作者一向推崇简洁编程理念，所以很长一段时间 Python 都没有三元操作符这么个概念（因为 Python 觉得三元操作符使得程序结构变复杂了），但是 Python 社区的小伙伴们表达了极大的诉求，所以最终 Python 的作者为 Python 加入了这个三元操作符。有了这个三元操作符的条件表达式，你可以使用一条语句来完成以下的条件判断和赋值操作：

```
if x < y:
    small = x
else:
    small = y
```

那么将上边的代码用传说中的三元操作符表示应该是怎样的呢？

三元操作符语法：

```
a = x if 条件 else y
```

表示当条件为 True 的时候，a 的值赋值为 x，否则赋值为 y。

所以，上面的例子可以改进为：

```
small = x if x < y else y
```

## 4.6 断言

断言（assert）的语法其实有点像是 if 条件分支语句的“近亲”，所以就放在一块来讲了。assert 这个关键字称为“断言”，当这个关键字后边的条件为假的时候，程序自动崩溃并抛出 AssertionError 的异常。

什么情况下需要这样的代码呢？当我们在测试程序的时候就很好用，因为与其让错误的条件导致程序今后莫名其妙地崩溃，不如在错误条件出现的那一瞬间实现“自我毁灭”：

```
>>> assert 3 < 4
>>> assert 3 > 4
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    assert 3 > 4
AssertionError
```

一般来说，可以用它在程序中置入检查点，当需要确保程序中的某个条件一定为真才能让程序正常工作时，assert 关键字就非常有用了。

## 4.7 while 循环语句



Python 的 while 循环跟 if 条件分支类似，在条件为真的情况下，执行一段代码，不同的是，只要条件为真，while 循环会一直重复执行那段代码，把这段代码称为循环体。

```
while 条件:
    循环体
```

## 4.8 for 循环语句

那么接下来谈谈 Python 的计数器循环，也就是 for 循环。虽然说 Python 是由 C 语言编写而来的，但是它的 for 循环跟 C 语言的 for 循环不太一样，Python 的 for 循环显得更为智能和强大！这主要表现在它会自动调用迭代器的 next()方法[①]，会自动捕获 StopIteration 异常并结束循环，所以这更像是一个具有现代化气质的 for 循环结构。

一起来实验一下：

```
>>> favourite = "FishC"
>>> for each in favourite:
    print(each, end = '')

FishC
```

## 4.9 range()

for 循环其实还有一个小伙伴：range()内建函数。

语法：

```
range( [start,] stop[, step = 1] )
```

这个 BIF 有三个参数，其中用中括号括起来的两个表示这两个参数是可选的。step＝1 表示第三个参数的默认值是 1。零基础的朋友因为还没有学到函数，对于参数这个概念可能不理解，没事，暂时你就认为是函数的装备，函数有了装备攻击力什么的都会相应增加。

range 这个 BIF 的作用是生成一个从 start 参数的值开始，到 stop 参数的值结束的数字序列。常与 for 循环混迹于各种计数循环之间。

只传递一个参数的 range()，例如 range(5)，它会将第一个参数默认设置为 0，生成 0～5 的所有数字(注：包含 0 但不包含 5)。

```
>>> for i in range(5):
    print(i)
0
1
2
3
4
```

传递两个参数的 range()：

```
>>> for i in range(2, 9):
    print(i)
2
```

① 在对象中的函数称为方法。

```
3
4
5
6
7
8
```

传递三个参数的 range()：

```
>>> for i in range(1, 10, 2):
    print(i)
1
3
5
7
9
```

range()可以说是跟 for 循环最适合做搭档的小伙伴，当然，for 循环魅力很大，它还有其他各式各样的小伙伴配合实现各种乱七八糟的功能，这个在讲解列表和元组的时候再介绍给大家吧。

## 4.10　break 语句

break 语句的作用是终止当前循环，跳出循环体。举个例子：

```
# p4_4.py
bingo = '小甲鱼是帅哥'
answer = input('请输入小甲鱼最想听的一句话：')
while True:
    if answer == bingo:
        break
    answer = input('抱歉，错了，请重新输入(答案正确才能退出游戏)：')
print('哎哟，帅哦～')
print('您真是小甲鱼肚子里的蛔虫啊^_^')
```

程序运行后，只有当用户输入“小甲鱼是帅哥”的时候，才会执行 break 语句，也就是跳出 while 循环体：

```
>>>
请输入小甲鱼最想听的一句话：小甲鱼是笨蛋
抱歉，错了，请重新输入(答案正确才能退出游戏)：小甲鱼是帅哥
哎哟，帅哦～
您真是小甲鱼肚子里的蛔虫啊^_^
>>>
```

## 4.11　continue 语句

continue 语句的作用是终止本轮循环并开始下一轮循环（这里要注意的是：在开始下一轮循环之前，会先测试循环条件）。举个例子：

```
# p4_5.py
for i in range(10):
    if i % 2 != 0:
        print(i)
        continue
    i += 2
    print(i)
```

大家不妨在不运行程序的情况下目测一下这个程序会打印出什么？

# 第5章 列表、元组和字符串

## 5.1 列表：一个“打了激素”的数组



有时候需要把一堆东西暂时存储起来，因为它们有某种直接或者间接的联系，需要把它们放在某种“组”或者“集合”中，因为将来可能用得上。很多接触过编程的朋友都知道或者听说过数组。数组这个概念呢，就是把一大堆同种类型的数据挨个儿摆在一块儿，然后通过数组下标进行索引。但数组有一个基本要求，就是你所放在一起的数据必须类型一致。由于Python的变量没有数据类型，也就是说，Python是没有数组的。但是呢，Python加入了更为强大的列表。

Python的列表有多强大？如果把数组比作是一个集装箱的话，那么Python的列表就是一个工厂的仓库了。列表真的非常有用，基本上所有的Python程序都要使用到列表，包括之前的打飞机游戏，里边的小飞机可以全部扔到一个列表中统一管理。

### 5.1.1 创建列表

创建列表和创建普通变量一样，用中括号括起一堆数据就可以了，数据之间用逗号隔开，这样一个普普通通的列表就创建成功了：

```
>>> number = [1, 2, 3, 4, 5]
```

我们说列表是打了激素的数组不是没有道理的，可以创建一个鱼龙混杂的列表：

```
>>> mix = [1, "小甲鱼", 3.14, [1, 2, 3]]
```

可以看到上边这个列表里有整型、字符串、浮点型数据，甚至还可以包含着另一个列表。当然，如果实在想不到要往列表里边塞什么数据的时候，可以先创建一个空列表：

```
>>> empty = []
```

### 5.1.2 向列表添加元素

列表相当灵活，所以它的内容不可能总是固定的，现在就来教大家如何向列表添加元素吧。要向列表添加元素，可以使用append()方法：

```
>>> number = [1, 2, 3, 4, 5]
>>> number.append(6)
```

```
>>> number
[1, 2, 3, 4, 5, 6]
```

可以看到，参数6已经被添加到列表number的末尾了。有读者可能会问，这个方法调用怎么跟平时的BIF内置函数调用不一样呢？嗯，因为append()不是一个BIF，它是属于列表对象的一个方法。

中间这个"."，大家暂时可以理解为范围的意思：append()这个方法是属于一个叫作number的列表对象的。关于对象的知识，咱暂时只需要理解这么多，后边再给大家介绍对象。同理，我们可以把数字7和8添加进去，但是我们发现似乎不能用append()同时添加多个元素：

```
>>> number.append(7, 8)
Traceback (most recent call last):
  File "<pyshell#122>", line 1, in <module>
    number.append(7, 8)
TypeError: append() takes exactly one argument (2 given)
```

这时候就可以使用extend()方法向列表末尾添加多个元素：

```
>>> number.extend(7, 8)
Traceback (most recent call last):
  File "<pyshell#123>", line 1, in <module>
    number.extend(7, 8)
TypeError: extend() takes exactly one argument (2 given)
```

哎呀，怎么又报错了呢?! 嗯，其实小甲鱼是故意的。extend()方法事实上使用一个列表来扩展另一个列表，所以它的参数应该是一个列表：

```
>>> number.extend([7, 8])
>>> number
[1, 2, 3, 4, 5, 6, 7, 8]
```

好，我们又再一次向世界证明我们成功了！但是又发现了一个问题，到目前为止，我们都是往列表的末尾添加数据，那如果我想"插队"呢？

当然没问题，想要往列表的任意位置插入元素，就要使用insert()方法。insert()方法有两个参数：第一个参数代表在列表中的位置，第二个参数是在这个位置处插入一个元素。不妨来试一下，让数字0出现在列表number的最前边：

```
>>> number.insert(1, 0)
>>> number
[1, 0, 2, 3, 4, 5, 6, 7, 8]
```

等等，不是说好插入到第一个位置吗？怎么插入后0还是排在1的后边呢？其实是这样的：凡是顺序索引，Python均从0开始，同时这也是大多数编程语言约定俗成的规范。那么大家知道为什么要用0来表示第一个数吗？

是因为计算机本身就是二进制的，在二进制的世界里只有两个数：0和1，当然，0就是二进制里的第一个数了，所以嘛，秉承着这样的传统，0也就习惯用于表示第一个数。所以，正确的做法应该是：

```
>>> number = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> number.insert(0, 0)
>>> number
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

## 5.1.3　从列表中获取元素



跟数组一样，可以通过元素的索引值(index)从列表获取单个元素，注意，列表索引值是从0开始的：

```
>>> name = ["鸡蛋", "鸭蛋", "鹅蛋", "李狗蛋"]
>>> name[0]
'鸡蛋'
>>> name[3]
'李狗蛋'
```

那按照这个方法让“李狗蛋”和“鸭蛋”的位置互调：

```
>>> name[1], name[3] = name[3], name[1]
>>> name
['鸡蛋', '李狗蛋', '鹅蛋', '鸭蛋']
```

## 5.1.4　从列表删除元素

从列表删除元素，这里也介绍三种方法：remove()、del和pop()。先演示一下用remove()删除元素：

```
>>> name.remove("李狗蛋")
>>> name
['鸡蛋', '鹅蛋', '鸭蛋']
```

使用remove()删除元素，你并不需要知道这个元素在列表中的具体位置，只需要知道该元素存在列表中就可以了。如果要删除的东西根本不在列表中，程序就会报错：

```
>>> name.remove("陈鸭蛋")
Traceback (most recent call last):
  File "<pyshell#138>", line 1, in <module>
    name.remove("陈鸭蛋")
ValueError: list.remove(x): x not in list
```

remove()方法并不能指定删除某个位置的元素，这时就要用del来实现：

```
>>> del name[1]
>>> name
['鸡蛋', '鸭蛋']
```

注意，del是一个语句，不是一个列表的方法，所以你不必在它后边加上小括号()。另外，如果你想删除整个列表，还可以直接用del加列表名删除：

```
>>> del name
>>> name
Traceback (most recent call last):
  File "<pyshell#142>", line 1, in <module>
```

```
    name
NameError: name 'name' is not defined
```

最后，演示用 pop()方法“弹出”元素：

```
>>> name = ["鸡蛋", "鸭蛋", "鹅蛋", "李狗蛋"]
>>> name.pop()
'李狗蛋'
>>> name.pop()
'鹅蛋'
>>> name
['鸡蛋', '鸭蛋']
```

大家看到了，pop()方法默认是弹出列表中的最后一个元素。但这个 pop()方法其实还可以灵活运用，当你为它加上一个索引值作为参数的时候，它会弹出这个索引值对应的元素：

```
>>> name = ["鸡蛋", "鸭蛋", "鹅蛋", "李狗蛋"]
>>> name.pop(2)
'鹅蛋'
>>> name
['鸡蛋', '鸭蛋', '李狗蛋']
```

### 5.1.5 列表分片

利用索引值，每次可以从列表获取一个元素，但是人总是贪心的，如果需要一次性获取多个元素，有没有办法实现呢？利用列表分片(slice)，可以方便地实现这个要求：

```
>>> name = ["鸡蛋", "鸭蛋", "鹅蛋", "李狗蛋"]
>>> name[0:2]
['鸡蛋', '鸭蛋']
```

很简单对吧？只不过是用一个冒号隔开两个索引值，左边是开始位置，右边是结束位置。这里要注意的一点是，结束位置上的元素是不包含的。利用列表分片，得到一个原来列表的拷贝，原来列表并没有发生改变。

列表分片也可以简写，我们说过 Python 就是以简洁闻名于世，所以你能想到的“便捷方案”，Python 的作者以及 Python 社区的小伙伴们都已经想到了，并付诸实践，你要做的就是验证一下是否可行：

```
>>> name[:2]
['鸡蛋', '鸭蛋']
>>> name[1:]
['鸭蛋', '鹅蛋', '李狗蛋']
>>> name[:]
['鸡蛋', '鸭蛋', '鹅蛋', '李狗蛋']
```

如果没有开始位置，Python 会默认开始位置是 0。同样道理，如果要得到从指定索引值到列表末尾的所有元素，把结束位置省去即可。如果没有放入任何索引值，而只有一个冒号，将得到整个列表的拷贝。

再一次强调：列表分片就是建立原列表的一个拷贝(或者说副本)，所以如果你想对列表做出某些修改，但同时还想保持原来的那个列表，那么直接使用分片的方法来获取拷贝就很方便了。

## 5.1.6　列表分片的进阶玩法

分片操作实际上还可以接收第三个参数，其代表的是步长，默认情况下（不指定它的时候）该值为1，来试试将其改成2会有什么效果？

```
>>> list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list1[0:9:2]
[1, 3, 5, 7, 9]
```

如果将步长改成2，那么每前进两个元素才取一个出来。其实还可以直接写成list1[::2]。如果步长的值是负数，例如－1，结果会怎样呢？不妨试试看：

```
>>> list1[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

是不是很有意思？这里步长设置为－1，就相当于复制一个反转的列表。

## 5.1.7　一些常用操作符

此前学过的大多数操作符都可以运用到列表上：

```
>>> list1 = [123]
>>> list2 = [234]
>>> list1 > list2
False
>>> list3 = ['abc']
>>> list4 = ['bcd']
>>> list3 < list4
True
```

我们发现列表还是挺聪明的，竟然会懂得比较大小。那如果列表中不止一个元素呢？结果又会如何？

```
>>> list1 = [123, 456]
>>> list2 = [234, 123]
>>> list1 > list2
False
```

怎么会这样？list1列表的和是123＋456＝579，按理应该比list2列表的和234＋123＝357要大，为什么list1＞list2还会返回False呢？

思考片刻后得出结论：Python的列表原来并没有我们想象中那么“智能”（注：在后边讲“魔法方法”的章节会教大家如何把列表改变得更加聪明），当列表包含多个元素的时候，默认是从第一个元素开始比较，只要有一个PK赢了，就算整个列表赢了。字符串比较也是同样的道理（字符串比较的是第一个字符对应的ASCII码值的大小）。

我们知道字符串可以用加号（＋）来进行拼接，用乘号（＊）来复制自身若干次。它们在列表身上也是可以实现的：

```
>>> list1 = [123, 456]
>>> list2 = [234, 123]
>>> list3 = list1 + list2
```

```
>>> list3
[123, 456, 234, 123]
```

加号(+)也叫连接操作符,它允许我们把多个列表对象合并在一起,其实就相当于extend()方法实现的效果。一般情况下建议大家使用extend()方法来扩展列表,因为这样显得更为规范和专业。另外,连接操作符并不能实现列表添加新元素的操作:

```
>>> list1 = [123, 456]
>>> list2 = list1 + 789
Traceback (most recent call last):
  File "<pyshell#177>", line 1, in <module>
    list2 = list1 + 789
TypeError: can only concatenate list (not "int") to list
```

所以如果要添加一个元素到列表中,用什么方法?嗯,可以用append()或者insert()方法,希望大家还记得。乘号(*)也叫重复操作符,重复操作符可以用于列表中:

```
>>> list1 = [123]
>>> list1 * 3
[123, 123, 123]
```

当然复合赋值运算符也可以用于列表:

```
>>> list1 *= 5
>>> list1
[123, 123, 123, 123, 123]
```

另外有个成员关系操作符大家也不陌生,我们是在谈for循环的时候认识它的,成员关系操作符就是in和not in!

```
>>> list1 = ["小猪", "小猫", "小狗", "小甲鱼"]
>>> "小甲鱼" in list1
True
>>> "小护士" not in list1
True
```

之前说过列表里边可以包含另一个列表,那么对于列表中的列表元素,能不能使用in和not in测试呢?试试便知:

```
>>> list1 = ["小猪", "小猫", ["小甲鱼", "小护士"], "小狗"]
>>> "小甲鱼" in list1
False
>>> "小护士" not in list1
True
```

可见,in和not in只能判断一个层次的成员关系,这跟break和continue语句只能作用于一个层次的循环是一个道理。那要判断列表里边的列表的元素,应该先进入一层列表:

```
>>> "小甲鱼" in list1[2]
True
>>> "小护士" not in list1[2]
False
```

顺便说一下,前面提到使用索引号去访问列表中的值,那么对于列表中的列表,应该如何访

问呢？

大家应该猜到了，其实跟C语言访问二维数组的方法相似：

```
>>> list1[2][0]
'小甲鱼'
```

### 5.1.8 列表的小伙伴们

接下来认识一下列表的小伙伴们，那么列表有多少小伙伴呢？不妨让Python自己告诉我们：

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__','__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

产生了一个熟悉又陌生的列表，很多熟悉的方法似曾相识，例如append()、extend()、insert()、pop()、remove()都是学过的。现在再给大家介绍几个常用的方法。

count()这个方法的作用是计算它的参数在列表中出现的次数：

```
>>> list1 = [1, 1, 2, 3, 5, 8, 13, 21]
>>> list1.count(1)
2
```

index()这个方法会返回它的参数在列表中的位置：

```
>>> list1.index(1)
0
```

可以看到，这里是返回第一个目标(1)在list1中的位置，index()方法还有两个参数，用于限定查找的范围。因此可以这样查找第二个目标在list1的位置：

```
>>> start = list1.index(1) + 1
>>> stop = len(list1)
>>> list1.index(1, start, stop)
1
```

reverse()方法的作用是将整个列表原地翻转，就是排最后的放到最前边，排最前的放到最后，那么排倒数第二的就排在第二，以此类推：

```
>>> list1 = [1, 2, 3, 4, 5, 6, 7, 8]
>>> list1.reverse()
>>> list1
[8, 7, 6, 5, 4, 3, 2, 1]
```

sort()这个方法是用指定的方式对列表的成员进行排序，默认不需要参数，从小到大排队：

```
>>> list1 = [8, 9, 3, 5, 2, 6, 10, 1, 0]
```

```
>>> list1.sort()
>>> list1
[0, 1, 2, 3, 5, 6, 8, 9, 10]
```

那如果需要从大到小排队呢？很简单，先调用 sort()方法，列表会先从小到大排好队，然后调用 reverse()方法原地翻转就可以啦。

什么？太麻烦？好吧，大家真是越来越懒了……很好，大家离天才又近了一步[①]。其实，sort()这个方法其实有三个参数，其形式为 sort(func, key, reverse)。

func 和 key 参数用于设置排序的算法和关键字，默认是使用归并排序，算法问题不在这里讨论，有兴趣的朋友可以看一下小甲鱼另一本不错的教程：《数据结构和算法》(C 语言)。这里要讨论 sort()方法的第三个参数：reverse，没错，就是刚刚学的那个 reverse()方法的那个 reverse。不过这里作为 sort()的一个默认参数，它的默认值是 sort(reverse=False)，表示不颠倒顺序。因此，只需要把 False 改为 True，列表就相当于从大到小排序：

```
>>> list1 = [8, 9, 3, 5, 2, 6, 10, 1, 0]
>>> list1.sort(reverse = True)
>>> list1
[10, 9, 8, 6, 5, 3, 2, 1, 0]
```

### 5.1.9 关于分片“拷贝”概念的补充

上一节提到使用分片创建列表的拷贝：

```
>>> list1 = [1, 3, 2, 9, 7, 8]
>>> list2 = list1[:]
>>> list2
[1, 3, 2, 9, 7, 8]
>>> list3 = list1
>>> list3
[1, 3, 2, 9, 7, 8]
```

看似一样，对吧？但事实上呢？利用列表的一个小伙伴做以下修改，大家看看差别：

```
>>> list1.sort()
>>> list1
[1, 2, 3, 7, 8, 9]
>>> list2
[1, 3, 2, 9, 7, 8]
>>> list3
[1, 2, 3, 7, 8, 9]
```

可以看到 list1 已经从小到大排好了序，那 list2 和 list3 呢？使用分片方式得到的 list2 很有原则、很有格调，并不会因为 list1 的改变而改变，这个原理我待会儿跟大家说；接着看 list3……看，真正的墙头草是 list3，它竟然跟着 list1 改变了，这是为什么呢？

不知道大家还记不记得在讲解变量的时候说过，Python 的变量就像一个标签，就一个名字而已……还是给大家画个图好理解，如图 5-1 所示。

---

① 小甲鱼个人认为“懒”是创新发明的根源和动力。

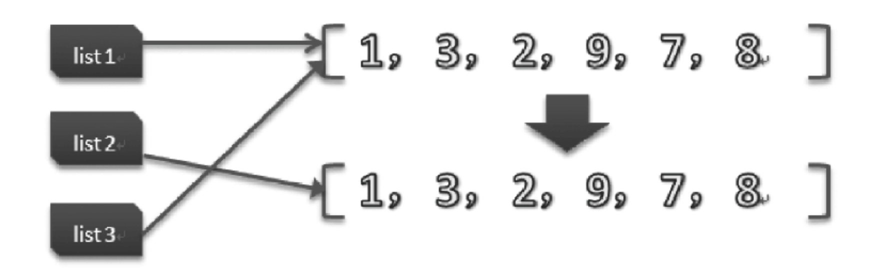



图 5-1 拷贝列表

这下大家应该明白了吧，为一个列表指定另一个名字的做法，只是向同一个列表增加一个新的标签而已，真正的拷贝是要使用分片的方法。这个也是初学者最容易混淆的地方，大家以后写代码时一定要注意哦。



## 5.2 元组：戴上了枷锁的列表

早在三百多年前，孟德斯鸠在《论法的精神》里边就提到"一切拥有权力的人都容易滥用权力，这是万古不变的一条经验。"但是凡是拥有大权力的人，都想用自身的实践证明孟德斯鸠是一个只会说屁话的家伙，但是他们好像都失败了……

由于列表过分强大，Python 的作者觉得这样似乎不妥，于是发明了列表的"表亲"——元组。

元组和列表最大的区别就是你可以任意修改列表中的元素，可以任意插入或者删除一个元素，而对元组是不行的，元组是不可改变的(像字符串一样)，所以你也别指望对元组进行原地排序等高级操作了。

### 5.2.1 创建和访问一个元组

元组和列表，除了不可改变这个显著特征之外，还有一个明显的区别是，创建列表用的是中括号，而创建元组大部分时候用的是小括号(注意，我这里说的是大部分)：

```
>>> tuple1 = (1, 2, 3, 4, 5, 6, 7, 8)
>>> tuple1
(1, 2, 3, 4, 5, 6, 7, 8)
```

访问元组的方式与列表无异：

```
>>> tuple1[1]
2
>>> tuple1[5:]
(6, 7, 8)
>>> tuple1[:5]
(1, 2, 3, 4, 5)
```

也使用分片的方式来复制一个元组：

```
>>> tuple2 = tuple1[:]
>>> tuple2
```

```
(1, 2, 3, 4, 5, 6, 7, 8)
```

如果你试图修改元组的一个元素，那么抱歉，Python会很不开心：

```
>>> tuple1[1] = 1
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    tuple1[1] = 1
TypeError: 'tuple' object does not support item assignment
```

我很好奇如果问你，列表的标志性符号是中括号([])，那么元组的标志性符号是什么？你会怎么回答呢？

小甲鱼相信百分之九十的朋友都会不假思索地回答：小括号啊，有部分比较激进的朋友还可能会补充一句“小甲鱼你傻啊？”

好吧，这个问题其实也是大部分初学者所忽略和容易上当的，我们实验一下：

```
>>> temp = (1)
>>> type(temp)
<class 'int'>
```

还记得type()方法吧，作用是返回参数的类型，这里它返回说temp变量是整型(int)。再试试：

```
>>> temp = 1, 2, 3
>>> type(temp)
<class 'tuple'>
```

噢，发现了吧？就算没有小括号，temp还是元组类型，所以逗号(,)才是关键，小括号只是起到补充的作用。但是你如果想要创建一个空元组，那么你就直接使用小括号即可：

```
>>> temp = ()
>>> type(temp)
<class 'tuple'>
```

所以这里要注意的是，如果要创建的元组中只有一个元素，请在它后边加上一个逗号(,)，这样可以明确告诉Python你要的是一个元组，不要拿什么整型、浮点型来忽悠你：

```
>>> temp1 = (1)
>>> type(temp1)
<class 'int'>
>>> temp2 = (1, )
>>> type(temp2)
<class 'tuple'>
>>> temp3 = 1,
>>> type(temp3)
<class 'tuple'>
```

为了证明逗号(,)起到了决定性作用，再给大家举个例子：

```
>>> 8 * (8)
64
>>> 8 * (8,)
(8, 8, 8, 8, 8, 8, 8, 8)
```



### 5.2.2 更新和删除元组

有朋友可能会说,刚才不是你自己说"元组是板上钉钉不能修改的吗"?你现在又来谈更新一个元组,小甲鱼你这不是自己打脸吗?

大家不要激动……我们只是讨论一个相对含蓄的做法(直接在同一个元组上更新是不可行的,除非你学习了后边的"魔法方法"章节)。不知道大家还记不记得以前是如何更新一个字符串的?没错,是通过拷贝现有的字符串片段构造一个新的字符串的方式解决的,对元组也是使用同样的方法:

```
>>> temp = ("小鸡", "小鸭", "小猪")
>>> temp = temp[:2] + ("小甲鱼",) + temp[2:]
>>> temp
('小鸡', '小鸭', '小甲鱼', '小猪')
```

上面的代码需要在"小鸭"和"小猪"中间插入"小甲鱼",那么通过分片的方法让元组拆分为两部分,然后再使用连接操作符(+)合并成一个新元组,最后将原来的变量名(temp)指向连接好的新元组。不妨可以把这样的做法称为"狸猫换太子"。在这里就要注意了,逗号是必需的,小括号也是必需的!

在谈到列表的时候,小甲鱼跟大家说有三个方法可以删除列表里边的元素,但是对于元组是不可变的原则来说,单独删除一个元素是不可能的,当然你可以用刚才小甲鱼教给大家更新元组的方法,间接地删除一个元素:

```
>>> temp = temp[:2] + temp[3:]
>>> temp
('小鸡', '小鸭', '小猪')
```

如果要删除整个元组,只要使用 del 语句即可显式地删除一个元组:

```
>>> del temp
>>> temp
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    temp
NameError: name 'temp' is not defined
```

其实在日常使用中,很少使用 del 去删除整个元组,因为 Python 的回收机制会在这个元组不再被使用到的时候自动删除。

最后小结一下哪些操作符可以使用在元组上,拼接操作符和重复操作符刚刚演示过了,关系操作符、逻辑操作符和成员关系操作符 in 和 not in 也可以直接应用在元组上,这跟列表是一样的,大家自己实践一下就知道了。关于列表和元组,我们今后会谈得更多,目前,就先聊到这里。

## 5.3 字符串



或许现在又回过头来谈字符串,有些朋友可能会觉得没必要。

其实关于字符串,还有很多你可能不知道的秘密,由于字符串在日常使用中是如此常见,

因此小甲鱼抱着负责任的态度在本节把所知道的都倒出来跟大家分享一下。

关于创建和访问字符串，前面已经介绍过了。不过学了列表和元组，我们知道了分片的概念，事实上也可以应用于字符串之上：

```
>>> str1 = "I love fishc.com!"
>>> str1[:6]
'I love'
```

接触过C语言的朋友应该知道，在C语言中，字符串和字符是两个不同的概念(C语言用单引号表示字符，双引号表示字符串)。但在Python并没有字符这个类型，在Python看来，所谓字符，就是长度为1的字符串。当要访问字符串的其中一个字符的时候，只需用索引列表或元组的方法来索引字符串即可：

```
>>> str1[5]
'e'
```

字符串跟元组一样，都是属于"一言既出、驷马难追"的家伙。所以一旦定下来就不能直接对它们进行修改了，如果必须要修改，我们就需要委曲求全……

```
>>> str1[:6] + " 插入的字符串" + str1[6:]
'I love 插入的字符串 fishc.com!'
```

但是大家要注意，这种通过拼接旧字符串的各个部分得到新字符串的方式并不是真正意义上的改变原始字符串，原来的那个"家伙"还在，只是将变量指向了新的字符串(旧的字符串一旦失去了变量的引用，就会被Python的垃圾回收机制释放掉)。

像比较操作符、逻辑操作符、成员关系操作符等的操作跟列表和元组是一样的，这里就不再啰唆了。

### 5.3.1 各种内置方法

列表和元组都有它们的方法，大家可能觉得列表的方法已经非常多了，其实字符串更多呢。表5-1总结了字符串的所有方法及对应的含义。

**表5-1 Python字符串的方法**

| 方　法 | 含　义 |
| --- | --- |
| capitalize() | 把字符串的第一个字符改为大写 |
| casefold() | 把整个字符串的所有字符改为小写 |
| center(width) | 将字符串居中，并使用空格填充至长度width的新字符串 |
| count(sub[, start[, end]]) | 返回sub在字符串里边出现的次数，start和end参数表示范围，可选 |
| encode(encoding='utf-8', errors='strict') | 以encoding指定的编码格式对字符串进行编码 |
| endswith(sub[, start[, end]]) | 检查字符串是否以sub子字符串结束，如果是返回True，否则返回False。start和end参数表示范围，可选 |
| expandtabs([tabsize=8]) | 把字符串中的tab符号(\t)转换为空格，如不指定参数，默认的空格数是tabsize=8 |
| find(sub[, start[, end]]) | 检测sub是否包含在字符串中，如果有则返回索引值，否则返回-1，start和end参数表示范围，可选 |



续表

| 方　　法 | 含　　义 |
|---|---|
| index(sub[, start[, end]]) | 跟 find 方法一样,不过如果 sub 不在 string 中会产生一个异常 |
| isalnum() | 如果字符串至少有一个字符并且所有字符都是字母或数字则返回 True,否则返回 False |
| isalpha() | 如果字符串至少有一个字符并且所有字符都是字母则返回 True,否则返回 False |
| isdecimal() | 如果字符串只包含十进制数字则返回 True,否则返回 False |
| isdigit() | 如果字符串只包含数字则返回 True,否则返回 False |
| islower() | 如果字符串中至少包含一个区分大小写的字符,并且这些字符都是小写,则返回 True,否则返回 False |
| isnumeric() | 如果字符串中只包含数字字符,则返回 True,否则返回 False |
| isspace() | 如果字符串中只包含空格,则返回 True,否则返回 False |
| istitle() | 如果字符串是标题化(所有的单词都是以大写开始,其余字母均小写),则返回 True,否则返回 False |
| isupper() | 如果字符串中至少包含一个区分大小写的字符,并且这些字符都是大写,则返回 True,否则返回 False |
| join(sub) | 以字符串作为分隔符,插入到 sub 中所有的字符之间 |
| ljust(width) | 返回一个左对齐的字符串,并使用空格填充至长度为 width 的新字符串 |
| lower() | 转换字符串中所有大写字符为小写 |
| lstrip() | 去掉字符串左边的所有空格 |
| partition(sub) | 找到子字符串 sub,把字符串分成一个 3 元组(pre_sub, sub, fol_sub),如果字符串中不包含 sub 则返回('原字符串', '', '') |
| replace(old, new[, count]) | 把字符串中的 old 子字符串替换成 new 子字符串,如果 count 指定,则替换不超过 count 次 |
| rfind(sub[, start[, end]]) | 类似于 find()方法,不过是从右边开始查找 |
| rindex(sub[, start[, end]]) | 类似于 index()方法,不过是从右边开始查找 |
| rjust(width) | 返回一个右对齐的字符串,并使用空格填充至长度为 width 的新字符串 |
| rpartition(sub) | 类似于 partition()方法,不过是从右边开始查找 |
| rstrip() | 删除字符串末尾的空格 |
| split(sep=None, maxsplit=-1) | 不带参数默认是以空格为分隔符切片字符串,如果 maxsplit 参数有设置,则仅分隔 maxsplit 个子字符串,返回切片后的子字符串拼接的列表 |
| splitlines(([keepends])) | 按照'\n'分隔,返回一个包含各行作为元素的列表,如果 keepends 参数指定,则返回前 keepends 行 |
| startswith(prefix[, start[, end]]) | 检查字符串是否以 prefix 开头,是则返回 True,否则返回 False。start 和 end 参数可以指定范围检查,可选 |
| strip([chars]) | 删除字符串前边和后边所有的空格,chars 参数可以定制删除的字符,可选 |
| swapcase() | 翻转字符串中的大小写 |
| title() | 返回标题化(所有的单词都是以大写开始,其余字母均小写)的字符串 |
| translate(table) | 根据 table 的规则(可以由 str.maketrans('a', 'b')定制)转换字符串中的字符 |
| upper() | 转换字符串中的所有小写字符为大写 |
| zfill(width) | 返回长度为 width 的字符串,原字符串右对齐,前边用 0 填充。 |

这里选几个常用的给大家演示一下用法，首先是 casefold()方法，它的作用是将字符串的所有字符变为小写：

```
>>> str1 = "FishC"
>>> str1.casefold()
'fishc'
```

count(sub[, start[, end]])方法之前试过了，就是查找 sub 子字符串出现的次数，可选参数(注：在 Python 文档中，用方括号([])括起来表示为可选)start 和 end 表示查找的范围：

```
>>> str1 = "AbcABCabCabcABCabc"
>>> str1.count('ab', 0, 15)
2
```

如果要查找某个子字符串在该字符串中的位置，可以使用 find(sub[, start[, end]])或 index(sub[, start[, end]])方法。如果找到了，则返回值是第一个字符的索引值；如果找不到，则 find()方法会返回－1，而 index()方法会抛出异常(注：异常是可以被捕获并处理的错误，目前你可以认为就是错误)：

```
>>> str1 = "I love fishc.com"
>>> str1.find("fishc")
7
>>> str1.find("good")
-1
>>> str1.index("fishc")
7
>>> str1.index("good")
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    str1.index("good")
ValueError: substring not found
```

今后你可能会在很多文档中看到 join(sub)的身影，程序员喜欢用它来连接字符串，但它的用法也许会让你感到诧异。join 是以字符串作为分隔符，插入到 sub 字符串中所有的字符之间：

```
>>> 'x'.join("Test")
'Txexsxt'
>>> '_'.join("FishC")
'F_i_s_h_C'
```

为什么说“程序员喜欢用 join()来连接字符串”，我们不是有很好用的连接符号(＋)吗？这是因为当使用连接符号(＋)去拼接大量的字符串时是非常低效率的，因为加号连接会引起内存复制以及垃圾回收操作。所以对于大量的字符串拼接来说，使用 join()方法的效率要高一些：

```
>>> 'I' + ' ' + 'love' + ' ' + 'fishc.com'
'I love fishc.com'
>>> ' '.join(['I', 'love', 'fishc.com'])
'I love fishc.com'
```

replace(old, new[, count])方法如其名，就是替换指定的字符串：

```
>>> str1 = "I love you"
>>> str1.replace("you", "fishc.com")
'I love fishc.com'
```

split(sep=None, maxsplit=-1)跟join()正好相反，split()用于拆分字符串：

```
>>> str1 = ' '.join(['I', 'love', 'fishc.com'])
>>> str1
'I love fishc.com'
>>> str1.split()
['I', 'love', 'fishc.com']
>>> str2 = '_'.join("FishC")
>>> str2.split(sep = '_')
['F', 'i', 's', 'h', 'C']
```

## 5.3.2 格式化

前面介绍了Python字符串大部分方法的使用，但唯独漏了一个format()方法。因为小甲鱼觉得format()方法跟本节的话题如出一辙，都是关于字符串的格式化，所以放一块来讲解。

那什么是字符串的格式化，又为什么需要对字符串进行格式化呢？举个小例子给大家听：某天小甲鱼召开了鱼C国际互联安全大会，到会的朋友有来自世界各地的各界精英人士，有小乌龟、喵星人、旺星人，当然还有米奇和唐老鸭。哇噻，那气势简直跟小甲鱼开了个动物园一样……但是问题来了，大家交流起来简直是鸡同鸭讲，不知所云！但是最后聪明的小甲鱼还是把问题给解决了，其实也很简单，各界都找一个翻译就行了，统一都翻译成普通话，那么问题就解决了……最后我们这个大会当然取得了卓越的成功并记入了吉尼斯动物大全。举这个例子就是想跟大家说，格式化字符串，就是按照统一的规格去输出一个字符串。如果规格不统一，就很可能造成误会，例如十六进制的10跟十进制的10或二进制的10完全是不同的概念（十六进制10等于十进制16，二进制10等于十进制2）。字符串格式化，正是帮助我们纠正并规范这类问题而存在的。

### 1. format()

format()方法接受位置参数和关键字参数（位置参数和关键字参数在函数章节有详细讲解），二者均传递到一个叫作replacement字段。而这个replacement字段在字符串内由大括号({})表示。先看一个例子：

```
>>> "{0} love {1}.{2}".format("I", "FishC", "com")
'I love FishC.com'
```

怎么回事呢？仔细看，字符串中的{0}、{1}和{2}应该跟位置有关，依次被format()的三个参数替换，那么format()的三个参数就叫作位置参数。那什么是关键字参数呢，再来看一个例子：

```
>>> "{a} love {b}.{c}".format(a = "I", b = "FishC", c = "com")
'I love FishC.com'
```

{a}、{b}和{c}就相当于三个标签，format()将参数中等值的字符串替换进去，这就是关键字参数啦。另外，你也可以综合位置参数和关键字参数在一起使用：

```
>>> "{0} love {b}.{c}".format("I", b = "FishC", c = "com")
'I love FishC.com'
```

但要注意的是，如果将位置参数和关键字参数综合在一起使用，那么位置参数必须在关键字参数之前，否则就会出错：

```
>>> "{a} love {b}.{0}".format(a = "I", b = "FishC", "com")
SyntaxError: non - keyword arg after keyword arg
```

如果要把大括号打印出来，你有办法吗？没错，这跟字符串转义字符有点像，只需要用多一层大括号包起来即可(要打印转义字符(\)，只需用转义字符转义本身(\\))：

```
>>> "{{0}}".format("不打印")
'{0}'
```

位置参数"不打印"没有被输出，这是因为{0}的特殊功能被外层的大括号({})剥夺，因此没有字段可以输出。注意，这并不会产生错误哦。最后来看另一个例子：

```
>>> "{0}: {1:.2f}".format("圆周率", 3.14159)
'圆周率: 3.14'
```

可以看到，位置参数{1}跟平常有些不同，后边多了个冒号。在替换域中，冒号表示格式化符号的开始，".2"的意思是四舍五入到保留两位小数点，而 f 的意思是浮点数，所以按照格式化符号的要求打印出了 3.14。

### 2. 格式化操作符：%

刚才讲的是字符串的格式化方法，现在来谈谈字符串所独享的一个操作符：%，有人说，这不是求余数的操作符吗？是的，没错。当%的左右均为数字的时候，那么它表示求余数的操作；但当它出现在字符中的时候，它表示的是格式化操作符。表 5-2 列举了 Python 的格式化符号及含义。

**表 5-2 Python 格式化符号及含义**

| 符　　号 | 含　　义 |
|---|---|
| %c | 格式化字符及其 ASCII 码 |
| %s | 格式化字符串 |
| %d | 格式化整数 |
| %o | 格式化无符号八进制数 |
| %x | 格式化无符号十六进制数 |
| %X | 格式化无符号十六进制数(大写) |
| %f | 格式化浮点数字，可指定小数点后的精度 |
| %e | 用科学计数法格式化浮点数 |
| %E | 作用同%e，用科学计数法格式化浮点数 |
| %g | 根据值的大小决定使用%f 或%e |
| %G | 作用同%g，根据值的大小决定使用%f 或%E |

下面给大家举几个例子参考：

```
>>> '%c' % 97
'a'
>>> '%c%c%c%c%c' % (70, 105, 115, 104, 67)
'FishC'
>>> '%d转换为八进制是：%o' % (123, 123)
'123转换为八进制是：173'
>>> '%f用科学计数法表示为：%e' % (149500000, 149500000)
'149500000.000000用科学计数法表示为：1.495000e+08'
```

Python还提供了格式化操作符的辅助指令，如表5-3所示。

**表5-3　格式化操作符的辅助指令**

| 符　　号 | 含　　义 |
|---|---|
| m.n | m是显示的最小总宽度，n是小数点后的位数 |
| - | 结果左对齐 |
| + | 在正数前面显示加号(+) |
| # | 在八进制数前面显示'0o'，在十六进制数前面显示'0x64'或'0X64' |
| 0 | 显示的数字前面填充'0'代替空格 |

同样给大家举几个例子供参考：

```
>>> '%5.1f' % 27.658
' 27.7'
>>> '%.2e' % 27.658
'2.77e+01'
>>> '%10d' % 5
'         5'
>>> '%-10d' % 5
'5         '
>>> '%010d' % 5
'0000000005'
>>> '%#X' % 100
'0X64'
```

### 3. Python的转义字符及含义

Python的部分转义字符已经使用了一段时间，是时候来给它做个总结了，见表5-4。

**表5-4　转义字符及含义**

| 符　　号 | 说　　明 | 符　　号 | 说　　明 |
|---|---|---|---|
| \' | 单引号 | \r | 回车符 |
| \" | 双引号 | \f | 换页符 |
| \a | 发出系统响铃声 | \o | 八进制数代表的字符 |
| \b | 退格符 | \x | 十六进制数代表的字符 |
| \n | 换行符 | \0 | 表示一个空字符 |
| \t | 横向制表符(TAB) | \\ | 反斜杠 |
| \v | 纵向制表符 | | |



## 5.4 序列

聪明的你可能已经发现，小甲鱼把列表、元组和字符串放在一块儿来讲解是有道理的，因为它们之间有很多共同点：

- 都可以通过索引得到每一个元素。
- 默认索引值总是从 0 开始(当然灵活的 Python 还支持负数索引)。
- 可以通过分片的方法得到一个范围内的元素的集合。
- 有很多共同的操作符(重复操作符、拼接操作符、成员关系操作符)。

我们把它们统称为：序列！下面介绍一些关于序列的常用 BIF(内建方法)。

### 1. list([iterable])

list()方法用于把一个可迭代对象转换为列表，很多朋友经常听到“迭代”这个词，但要是让你解释的时候，很多朋友就含糊其词了：迭代……迭代就是 for 循环嘛……

这里小甲鱼帮大家科普一下：所谓迭代，是重复反馈过程的活动，其目的通常是为了接近并到达所需的目标或结果。每一次对过程的重复被称为一次“迭代”，而每一次迭代得到的结果会被用来作为下一次迭代的初始值……就目前来说，迭代还就是一个 for 循环，但今后会介绍到迭代器，那个功能，那叫一个惊艳！

好了，这里说 list()方法要么不带参数，要么带一个可迭代对象作为参数，而这个序列天生就是可迭代对象(迭代这个概念实际上就是从序列中泛化而来的)。还是通过几个例子给大家讲解吧：

```
>>> # 创建一个空列表
>>> a = list()
>>> a
[]
>>> # 将字符串的每个字符迭代存放到列表中
>>> b = list("FishC")
>>> b
['F', 'i', 's', 'h', 'C']
>>> # 将元组中的每个元素迭代存放到列表中
>>> c = list((1, 1, 2, 3, 5, 8, 13))
>>> c
[1, 1, 2, 3, 5, 8, 13]
```

事实上这个 list()方法大家自己也可以动手实现对不对？很简单嘛，实现过程大概就是新建一个列表，然后循环通过索引迭代参数的每一个元素并加入列表，迭代完毕后返回列表即可。大家课后不妨自己动动手来尝试一下。

### 2. tuple([iterable])

tuple()方法用于把一个可迭代对象转换为元组，具体的用法和 list()一样，这里就不啰唆了。

### 3. str(obj)

str()方法用于把 obj 对象转换为字符串，这个方法在前面结合 int()和 float()方法给大家

讲过,还记得吧?

### 4. len(sub)

len()方法用于返回 sub 参数的长度:

```
>>> str1 = "I love fishc.com"
>>> len(str1)
16
>>> list1 = [1, 1, 2, 3, 5, 8, 13]
>>> len(list1)
7
>>> tuple1 = "这", "是", "一", "个", "元组"
>>> len(tuple1)
5
```

### 5. max(…)

max()方法用于返回序列或者参数集合中的最大值,也就是说,max()的参数可以是一个序列,返回值是该序列中的最大值;也可以是多个参数,那么 max()将返回这些参数中最大的一个:

```
>>> list1 = [1, 18, 13, 0, -98, 34, 54, 76, 32]
>>> max(list1)
76
>>> str1 = "I love fishc.com"
>>> max(str1)
'v'
>>> max(5, 8, 1, 13, 5, 29, 10, 7)
29
```

### 6. min(…)

min()方法跟 max()用法一样,但效果相反:返回序列或者参数集合中的最小值。这里需要注意的是,使用 max()方法和 min()方法都要保证序列或参数的数据类型统一,否则会出错。

```
>>> list1 = [1, 18, 13, 0, -98, 34, 54, 76, 32]
>>> list1.append("x")
>>> max(list1)
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    max(list1)
TypeError: unorderable types: str() > int()
>>> min(123, 'oo', 456, 'xx')
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    min(123, 'oo', 456, 'xx')
TypeError: unorderable types: str() < int()
```

人家说:外行看热闹,内行看门道。分析一下这个错误信息。Python 这里说"TypeError:

unorderable types: str() > int()”，意思是说不能拿字符串和整型进行比较。这说明了什么呢？

你看，str() > int()，说明 max()方法和 min()方法的内部实现事实上类似于之前提到的，通过索引得到每一个元素，然后将各个元素进行对比。所以不妨根据猜想写出可能的代码：

```
# 猜想下 max(tuple1)的实现方式
temp = tuple1[0]

for each in tuple1:
    if each > temp:
        temp = each

return temp
```

由此可见，Python 的内置方法其实也没啥了不起的，有些我们也可以自己实现，对吧？所以只要认真跟着本书的内容学习下去，很多看似如狼似虎的问题，将来都能迎刃而解！

### 7. sum(iterable[, start])

sum()方法用于返回序列 iterable 的总和，用法跟 max()和 min()一样。但 sum()方法有一个可选参数(start)，如果设置该参数，表示从该值开始加起，默认值是 0：

```
>>> tuple1 = 1, 2, 3, 4, 5
>>> sum(tuple1)
15
>>> sum(tuple1, 10)
25
```

### 8. sorted(iterable, key=None, reverse=False)

sorted()方法用于返回一个排序的列表，大家还记得列表的内建方法 sort()吗？它们的实现效果一致，但列表的内建方法 sort()是实现列表原地排序；而 sorted()是返回一个排序后的新列表。

```
>>> list1 = [1, 18, 13, 0, -98, 34, 54, 76, 32]
>>> list2 = list1[:]
>>> list1.sort()
>>> list1
[-98, 0, 1, 13, 18, 32, 34, 54, 76]
>>> sorted(list2)
[-98, 0, 1, 13, 18, 32, 34, 54, 76]
>>> list2
[1, 18, 13, 0, -98, 34, 54, 76, 32]
```

### 9. reversed(sequence)

reversed()方法用于返回逆向迭代序列的值。同样的道理，实现效果跟列表的内建方法 reverse()一致。区别是列表的内建方法是原地翻转，而 reversed()是返回一个翻转后的迭代

器对象。你没看错，它不是返回一个列表，是返回一个迭代器对象：

```
>>> list1 = [1, 18, 13, 0, -98, 34, 54, 76, 32]
>>> reversed(list1)
<list_reverseiterator object at 0x000000000324F518>
>>> for each in reversed(list1):
	print(each, end=',')

32,76,54,34,-98,0,13,18,1,
```

### 10. enumerate(iterable)

enumerate()方法生成由二元组（二元组就是元素数量为二的元组）构成的一个迭代对象，每个二元组是由可迭代参数的索引号及其对应的元素组成的。举个例子你就明白了：

```
>>> str1 = "FishC"
>>> for each in enumerate(str1):
	print(each)

(0, 'F')
(1, 'i')
(2, 's')
(3, 'h')
(4, 'C')
```

### 11. zip(iter1 [,iter2 [...]])

zip()方法用于返回由各个可迭代参数共同组成的元组，举个例子比较容易理解：

```
>>> list1 = [1, 3, 5, 7, 9]
>>> str1 = "FishC"
>>> for each in zip(list1, str1):
	print(each)

(1, 'F')
(3, 'i')
(5, 's')
(7, 'h')
(9, 'C')
>>> tuple1 = (2, 4, 6, 8, 10)
>>> for each in zip(list1, str1, tuple1):
	print(each)

(1, 'F', 2)
(3, 'i', 4)
(5, 's', 6)
(7, 'h', 8)
(9, 'C', 10)
```




# 函数



## 6.1 Python的乐高积木

小时候大家应该都玩过乐高积木，只要通过想象和创意，可以用它拼凑出很多神奇的东西。随着学习的深入，编写的代码日益增加并且越来越复杂，所以需要找一个方法对这些复杂的代码进行重新组织。这么做的目的是使得代码更为简单易懂。我们说优秀的东西永远是经典的，而经典的东西永远是简单的。不是说复杂不好，要能够把复杂的东西简单化才能成为经典。

为了使得程序的代码更为简单，就需要把程序分解成较小的组成部分。这里会教大家三种方法来实现，分别是函数、对象和模块。

### 6.1.1 创建和调用函数

函数就是把代码打包成不同形状的乐高积木，以便可以发挥想象力进行随意拼装和反复使用。此前接触的BIF就是Python帮我们封装好的函数，用的时候很方便，根本不需要去想实现的原理，这就是把复杂变简单。

因为这几部分内容奠定了Python编程者的基本功底，所以小甲鱼在这几部分的准备上是花足了心思的，大家不要嫌啰唆——经常变着花样儿重复出现的内容肯定是最重要的。

简单来讲，一个程序可以按照不同功能的实现，分割成许许多多的代码块，每一个代码块就可以封装成一个函数。在Python中创建一个函数用def关键字：

```
>>> def myFirstFunction():
	print("这是我创建的第一个函数!")
	print("我表示很激动…")
	print("在这里,我要感谢TBB,感谢CCAV!")
```

注意，在函数名后边要加上一对小括号哦。这对小括号是必不可少的，因为有时候需要在里边放点东西，至于放什么，小甲鱼先卖个关子，待会儿告诉你。

我们创建了一个函数，但是从来都不去调用它，那么这个函数里的代码就永远也不会被执行。这里教大家如何调用一个函数。调用一个函数也非常简单，直接写出函数名加上小括号即可：

```
>>> myFirstFunction()
这是我创建的第一个函数!
```

```
我表示很激动…
在这里,我要感谢 TBB,感谢 CCAV!
```

函数的调用和运行机制：当函数 myFirstFunction()发生调用操作的时候，Python 会自动往上找到 def myFirstFunction()的定义过程，然后依次执行该函数所包含的代码块部分（也就是冒号后边的缩进部分内容）。只需要一条语句，就可以轻松地实现函数内的所有功能。假如我想把刚才的内容打印 3 次，我只需要调用 3 次函数即可：

```
>>> for i in range(3):
    myFirstFunction()

这是我创建的第一个函数!
我表示很激动…
在这里,我要感谢 TBB,感谢 CCAV!
这是我创建的第一个函数!
我表示很激动…
在这里,我要感谢 TBB,感谢 CCAV!
这是我创建的第一个函数!
我表示很激动…
在这里,我要感谢 TBB,感谢 CCAV!
```

## 6.1.2　函数的参数

现在可以来谈谈括号里是什么东西了！其实括号里放的就是函数的参数。在函数刚开始被发明出来的时候，是没有参数的（也就是说，小括号里没有内容），很快就引来了许多小伙伴们的质疑：函数不过是对做同样内容的代码进行打包，这样跟使用循环就没有什么本质不同了。

所以，为了使每次调用的函数可以有不同的实现，加入了参数的概念。例如，你封装了一个开炮功能的函数，默认武器是大炮，那用来打飞机是没问题的，但是你如果用这个函数来打小鸟，除非是愤怒的小鸟，否则就有点奇葩了。有了参数的实现，就可以轻松地将大炮换成步枪。总而言之，参数就是使得函数可以实现个性化：

```
>>> def mySecondFunction(name):
    print(name + "是帅锅!")

>>> mySecondFunction("小甲鱼")
小甲鱼是帅锅!
>>> mySecondFunction("小鱿鱼")
小鱿鱼是帅锅!
>>> mySecondFunction("小丑鱼")
小丑鱼是帅锅!
```

刚才的例子只有一个参数，使用多个参数，只需要用逗号隔开即可：

```
>>> def add(num1, num2):
    print(num1 + num2)

>>> add(1, 2)
3
```

那有些读者要问了，到底 Python 的函数支持多少参数呢？实际上你想要有多少个参数就可以有多少个参数，就像 Windows 的某些 API 函数就有十几个参数。但是建议大家自己定义的函数参数尽量不要太多，函数的功能和参数的意义也要相应写好注释，这样别人来维护你的程序才不会那么费劲！

### 6.1.3 函数的返回值

有些时候，需要函数为我们返回一些数据来报告执行的结果，譬如刚才提到具有开炮功能的函数，炮弹发射了之后到底是打中了没有？你总得有个交代吧。所以，我们的函数需要返回值。其实也非常简单，只需要在函数中使用关键字 return，后边跟着的就是指定要返回的值：

```
>>> def add(num1, num2):
    return num1 + num2

>>> add(1, 2)
3
```

## 6.2 灵活即强大



有时候，评论一种编程语言是否优秀，往往是看它是否灵活。灵活并非意味着无所不能、无所不包，那样就会显得庞大和冗杂。灵活应该表现为多变，比如前面学到的参数，函数因参数而灵活。如果没有参数，一个函数就只能死板地完成一个功能、一项任务。

### 6.2.1 形参和实参

参数从调用的角度来说，分为形式参数(parameter)和实际参数(argument)(注：本书后边简称为形参和实参)。跟绝大多数编程语言一样，形参指的是函数创建和定义过程中小括号里的参数，而实参则指的是函数在被调用的过程中传递进来的参数。举个例子：

```
>>> def myFirstFunction(name):
    print(name)

>>> myFirstFunction("小甲鱼")
小甲鱼
```

myFristFunction(name)的 name 是形参，因为它只是代表一个位置、一个变量名；而调用 myFirstFunction("小甲鱼")传递的"小甲鱼"是实参，因为它是一个具体的内容，是赋值到变量名中的值！

### 6.2.2 函数文档

给函数写文档是为了让别人可以更好地理解你的函数，所以这是一个好习惯。有些读者可能不理解为什么自己写的函数要跟别人分享呢？因为在实际开发中，个人的工作量和能力确实相当有限，因此中大型的程序永远都是团队来完成的。大家的代码要相互衔接，就需要先阅读别人提供的文档，因此适当的文档说明非常重要。而函数文档的作用是描述该函数的功

能，当然，这是写给人看的：

```
>>> def exchangeRate(dollar):
    """美元 -> 人民币
    汇率暂定为 6.5
    """
    return dollar * 6.5

>>> exchangeRate(10)
65.0
```

我们看到，在函数开头写下的字符串是不会打印出来的，但它会作为函数的一部分存储起来。这个称为函数文档字符串，它的功能跟注释是一样的。

那有读者可能会说，既然一样，搞那么复杂干啥呀？其实也不是完全一样，函数的文档字符串可以通过特殊属性__doc__获取（注：__doc__两边分别是两条下划线）：

```
>>> exchangeRate.__doc__
'美元 -> 人民币\n\t 汇率暂定为 6.5\n\t'
```

另外，想用一个函数却不确定其用法的时候，会通过 help()函数来查看函数的文档。因此，对我们自己的函数也可以依法炮制：

```
>>> help(exchangeRate)
Help on function exchangeRate in module __main__:

exchangeRate(dollar)
    美元 -> 人民币
    汇率暂定为 6.5
```

### 6.2.3　关键字参数

普通的参数叫位置参数，通常在调用一个函数的时候，粗心的程序员很容易会搞乱位置参数的顺序，以至于函数无法按照预期实现。因此，有了关键字参数。使用关键字参数，就可以很简单地解决这个潜在的问题，来看个例子：

```
>>> def saySomething(name, words):
    print(name + '->' + words)

>>> saySomething("小甲鱼", "让编程改变世界!")
小甲鱼->让编程改变世界!
>>> saySomething("让编程改变世界!", "小甲鱼")
让编程改变世界!->小甲鱼
>>> saySomething(words = "让编程改变世界!", name = "小甲鱼")
小甲鱼->让编程改变世界!
```

关键字参数其实就是在传入实参时指定形参的变量名，尽管使用这种技巧要多打一些字，但随着程序规模越来越大、参数越来越多，关键字参数起到的作用就越明显。毕竟宁可多打几个字符，也不希望出现料想不及的 BUG。

### 6.2.4　默认参数

初学者很容易搞混关键字参数和默认参数，默认参数是在定义的时候赋予了默认值的

参数：

```
>>> def saySomething(name = "小甲鱼", words = "让编程改变世界!"):
	print(name + '->' + words)

>>> saySomething()
小甲鱼->让编程改变世界!
>>> saySomething("苏轼", "不识庐山真面目,只缘身在此山中。")
苏轼->不识庐山真面目,只缘身在此山中。
>>> saySomething(words = "古之成大事者,不惟有超世之才,亦有坚忍不拔之志。", name = "苏轼")
苏轼->古之成大事者,不惟有超世之才,亦有坚忍不拔之志。
```

使用默认参数的话，就可以不带参数去调用函数。所以，它们之间的区别是：关键字参数是在函数调用的时候，通过参数名指定要赋值的参数，这样做就不怕因为搞不清参数的顺序而导致函数调用出错；而默认参数是在参数定义的过程中，为形参赋初值，当函数调用的时候，不传递实参，则默认使用形参的初始值代替。

### 6.2.5 收集参数

这个名字听起来比较新鲜，其实大多数时候它也被称作可变参数。发明这种机制的动机是函数的作者有时候也不知道这个函数到底需要多少个参数……听起来有点令人不解，但确实有这类情况。这时候，仅需要在参数前边加上星号(*)即可：

```
>>> def test(*params):
	print("有 %d 个参数" % len(params))
	print("第二个参数是:", params[1])
>>> test('F', 'i', 's', 'h', 'C')
有 5 个参数
第二个参数是: i
>>> test("小甲鱼", 123, 3.14)
有 3 个参数
第二个参数是: 123
```

其实大家仔细思考后也不难理解，Python 就是把标志为收集参数的参数们打包成一个元组。不过这里需要注意一下，如果在收集参数后边还需要指定其他参数，在调用函数的时候就应该使用关键参数来指定，否则 Python 就都会把你的实参都列入收集参数的范畴。举个例子：

```
>>> def test(*params, extra):
	print("收集参数是:", params)
	print("位置参数是:", extra)
>>> test(1, 2, 3, 4, 5, 6, 7, 8)
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    test(1, 2, 3, 4, 5, 6, 7, 8)
TypeError: test() missing 1 required keyword-only argument: 'extra'
>>> test(1, 2, 3, 4, 5, 6, 7, extra = 8)
收集参数是: (1, 2, 3, 4, 5, 6, 7)
位置参数是: 8
```

建议大家如果你的参数中带有收集参数，那么可将其他参数设置为默认参数，这样不容易出错：

```
>>> def test( * params, extra = "8"):
    print("收集参数是: ", params)
    print("位置参数是: ", extra)
>>> test(1, 2, 3, 4, 5, 6, 7, 8)
收集参数是: (1, 2, 3, 4, 5, 6, 7, 8)
位置参数是: 8
```

星号(*)其实既可以打包又可以“解包”。“解包”又是怎么回事呢？举个例子，假如你需要将一个列表a传入test参数的收集参数*param中，那么调用test(a)时便会出错，此时需要在a前边加上个星号(*)表示实参需要“解包”后才能使用：

```
>>> def test( * params):
    print("有 %d 个参数" % len(params))
    print("第二个参数是: ", params[1])
>>> a = [1, 2, 3, 4, 5, 6, 7, 8]
>>> test(a) # 直接将列表名 a 作为实参将会出错
有 1 个参数
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    test(a) # 直接将列表名 a 作为实参将会出错
  File "<pyshell#46>", line 3, in test
    print("第二个参数是: ", params[1])
IndexError: tuple index out of range
>>> test( * a) # 实参前边加上星号( * )表示解包
有 8 个参数
第二个参数是: 2
```

Python还有另一种收集方式，就是用两个星号(**)表示。跟前面的介绍不同，两个星号的收集参数表示为将参数们打包成字典的形式。字典的概念还没有接触，所以在后边讲解字典的章节中再给大家介绍吧。

## 6.3　我的地盘听我的



### 6.3.1　函数和过程

在很多编程语言中，函数和过程其实是区分开的。一般认为函数(function)是有返回值的，而过程(procedure)是简单、特殊并且没有返回值的。也就是说，函数是干完事儿必须写报告的“苦逼”，而过程是完事后拍拍屁股一走了之的“小混蛋”。

Python严格来说只有函数，没有过程！此话怎讲？有些朋友可能会说，在没有介绍return之前，Python的函数不也没有返回值吗？此言差矣，为了让大家更好地理解“Python严格来说只有函数，没有过程！”这句话，大家一起看看下面的例子：

```
>>> def hello():
    print("Hello~")
>>> print(hello())
Hello~
None
```

调用 print(hello())之后打印了两行文字，第一行我们当然知道是 hello()函数执行的，第二行的 None 是怎么回事？没错，大家猜对了。当不写 return 语句的时候，默认 Python 会认为函数是 return None 的。所以说 Python 所有的函数都有返回值。

### 6.3.2 再谈谈返回值

在许多编程语言中，我们说一个函数是整型，其实我们的意思是指这个函数会返回一个整型的返回值。而 Python 不这么干，Python 可以动态确定函数的类型，而且函数还能返回不同类型的值。还记得以前说过"Python 没有变量，只有名字"这句话吗？只需知道 Python 会返回一个东西，然后拿来用就可以了。另外，Python 似乎还可以同时返回多个值：

```
>>> def test():
    return [1, '小甲鱼', 3.14]
>>> test()
[1, '小甲鱼', 3.14]
```

Python 可以利用列表打包多种类型的值一次性返回。当然，你也可以直接用元组的形式返回多个值：

```
>>> def test():
    return 1, '小甲鱼', 3.14
>>> test()
(1, '小甲鱼', 3.14)
```

### 6.3.3 函数变量的作用域

其实这里要谈的是函数变量的作用域，也许你早已经听说了局部变量和全局变量，也许你早已经熟练于其他编程语言的变量作用域的细节，但无论如何这里你要认真学，因为这是重点，也许真有你平时注意不到的细节呢。

变量的作用域也就是平时所说的变量可见性，如上所说，一般的编程语言都有局部变量(Local Variable)和全局变量(Global Variable)之分。分析以下代码：

```
# p6_1.py
def discounts(price, rate):
    final_price = price * rate
    return final_price

old_price = float(input('请输入原价: '))
rate = float(input('请输入折扣率: '))
new_price = discounts(old_price, rate)
print('打折后价格是: ', new_price)
```

程序执行结果如下：

```
>>>
请输入原价: 80
请输入折扣率: 0.75
打折后价格是: 60.0
>>>
```

来分析一下代码：在函数 discounts()中，两个参数是 price 和 rate，还有一个是 final_price，它们都是 discounts()函数中的局部变量。为什么把它们称为局部变量呢？不妨修改一下代码：

```
# p6_2.py
def discounts(price, rate):
    final_price = price * rate
    return final_price

old_price = float(input('请输入原价: '))
rate = float(input('请输入折扣率: '))
new_price = discounts(old_price, rate)
print('打折后价格是: ', new_price)
print('这里试图打印局部变量 final_price 的值: ', final_price)
```

程序走起，像刚才一样输入之后程序便报错了：

```
>>>
请输入原价: 80
请输入折扣率: 0.75
打折后价格是: 60.0
Traceback (most recent call last):
  File "E:\p6_2.py", line 10, in <module>
    print('这里试图打印局部变量 final_price 的值: ', final_price)
NameError: name 'final_price' is not defined
>>>
```

错误原因：final_price 没有被定义过，也就是说，Python 找不到 final_price 这个变量。这是因为 final_price 只是一个局部变量，它的作用范围只在它的地盘上——discounts()函数的定义范围内——有效，出了这个范围，就不再属于它的地盘了，它将什么都不是。

总结一下：在函数里边定义的参数以及变量，都称为局部变量，出了这个函数，这些变量都是无效的。事实上的原理是，Python 在运行函数的时候，利用栈(Stack)进行存储，当执行完该函数后，函数中的所有数据都会被自动删除。所以在函数外边是无法访问到函数内部的局部变量的。

与局部变量相对的是全局变量，程序中 old_price、new_price、rate 都是在函数外边定义的，它们都是全局变量，全局变量拥有更大的作用域，例如在函数中可以访问到它们：

```
# p6_3.py
def discounts(price, rate):
    final_price = price * rate
    print('这里试图打印全局变量 old_price 的值: ', old_price)
    return final_price

old_price = float(input('请输入原价: '))
rate = float(input('请输入折扣率: '))
new_price = discounts(old_price, rate)
print('打折后价格是: ', new_price)
```

程序执行结果如下：

```
>>>
```

```
请输入原价：80
请输入折扣率：0.75
这里试图打印全局变量 old_price 的值：80.0
打折后价格是：60.0
>>>
```

真的可以实现，看上去似乎全局变量更为霸道！不过使用全局变量的时候要千万小心，在任何一种编程语言中都是如此。在 Python 中，你可以在函数中肆无忌惮地访问一个全局变量，但如果你试图去修改它，就会有奇怪的事情会发生了。分析下面的代码：

```
# p6_4.py
def discounts(price, rate):
    final_price = price * rate
    old_price = 50 # 这里试图修改全局变量
    print('在局部变量中修改后 old_price 的值是：', old_price)
    return final_price

old_price = float(input('请输入原价：'))
rate = float(input('请输入折扣率：'))
new_price = discounts(old_price, rate)
print('全局变量 old_price 现在的值是：', old_price)
print('打折后价格是：', new_price)
```

程序执行结果如下：

```
>>>
请输入原价：80
请输入折扣率：0.75
在局部变量中修改后 old_price 的值是：50
全局变量 old_price 现在的值是：80.0
打折后价格是：60.0
>>>
```

这里我就不吊大家的胃口了，如果在函数内部试图修改全局变量，那么 Python 会创建一个新的局部变量替代（名字跟全局变量相同），但真正的全局变量是纹丝不动的，所以实现的结果和大家的预期不同。

关于全局变量，我们也来总结一下：全局变量在整个代码段中都是可以访问到的，但是不要试图在函数内部去修改全局变量的值，因为那样 Python 会自动在函数内部新建一个名字一样的局部变量代替。

对于初学者来说，局部变量和全局变量在使用上很容易犯错，尤其是很多朋友都试图去建立一个跟全局变量同名的局部变量，这类做法小甲鱼是强烈反对的。那我如果想在函数里边去修改全局变量的值，有办法实现吗？如果我在函数里边想嵌套定义一个新的函数，可以吗？这些问题的答案都是肯定的，不过我们将在下一节再跟大家详细讲解。

## 6.4 内嵌函数和闭包



### 6.4.1 global 关键字

全局变量的作用域是整个模块（整个代码段），也就是代码段内所有的函数内部都可以访

问到全局变量。但要注意的一点是，在函数内部仅仅去访问全局变量就好，不要试图去修改它。

因为那样的话，Python 会使用屏蔽（Shadowing）的方式“保护”全局变量：一旦函数内部试图修改全局变量，Python 就会在函数内部自动创建一个名字一模一样的局部变量，这样修改的结果只会修改到局部变量，而不会影响到全局变量。看下面的例子：

```
>>> count = 5
>>> def myFun():
        count = 10
        print(count)

>>> myFun()
10
>>> count
5
```

但是毕竟人是要灵活应变的，假设你已经完全了解在函数中修改全局变量可能会导致程序可读性变差、出现莫名其妙的 BUG、代码的维护成本提高，但你还是坚持“虚心接受，死性不改”这八字原则，仍然觉得有必要在函数中去修改这个全局变量，那么你不妨可以使用 global 关键字来达到目的！修改程序如下：

```
>>> count = 5
>>> def myFun():
        global count
        count = 10
        print(count)

>>> myFun()
10
>>> count
10
```

### 6.4.2 内嵌函数

Python 的函数定义是可以嵌套的，也就是允许在函数内部创建另一个函数，这种函数叫作内嵌函数或者内部函数。接着看另一个例子：

```
>>> def fun1():
        print("fun1()正在被调用…")
        def fun2():
            print("fun2()正在被调用…")
        fun2()

>>> fun1()
fun1()正在被调用…
fun2()正在被调用…
```

这是函数嵌套的最简单的例子，虽然看起来没什么用……不过它麻雀虽小，五脏俱全。关于内部函数的使用，有一个比较值得注意的地方，就是内部函数整个作用域都在外部函数之

内。就像刚才例子中的 fun2()整个函数的作用域都在 fun1()里边。

需要注意的地方是，除了在 fun1()这个函数体中可以随意调用 fun2()这个内部函数外，出了 fun1()，就没有任何可以对 fun2()进行的调用。如果在 fun1()外部试图调用内部函数 fun2()，就会报错：

```
>>> fun2()
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    fun2()
NameError: name 'fun2' is not defined
```

### 6.4.3 闭包(closure)

闭包(closure)是函数式编程的一个重要的语法结构，函数式编程是一种编程范式，著名的函数式编程语言就是 LISP 语言(大家可能听说过这门语言，主要应用于绘图和人工智能，一直被认为是天才程序员使用的语言)。

那么不同的编程语言实现闭包的方式不同，Python 中的闭包从表现形式上定义为：如果在一个内部函数里，对在外部作用域(但不是在全局作用域)的变量进行引用，那么内部函数就被认为是闭包(closure)。还是来举个例子说明比较好理解：

```
>>> def funX(x):
    def funY(y):
        return x * y
    return funY

>>> i = funX(8)
>>> i(5)
40
```

也可以直接这么写：

```
>>> funX(8)(5)
40
```

通过上面的例子理解闭包的概念：如果在一个内部函数里(funY 就是这个内部函数)对外部作用域(但不是在全局作用域)的变量进行引用(x 就是被引用的变量，x 在外部作用域 funX 函数里面，但不在全局作用域里)，则这个内部函数(funY)就是一个闭包。

使用闭包需要注意的是：因为闭包的概念就是由内部函数而来的，所以你也不能在外部函数以外的地方对内部函数进行调用，下面的做法是错误的：

```
>>> funY(5)
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    funY(5)
NameError: name 'funY' is not defined
```

在闭包中，外部函数的局部变量对应内部函数的局部变量，事实上相当于之前讲的全局变量跟局部变量的对应关系，在内部函数中，你只能对外部函数的局部变量进行访问，但不能进行修改。

```
>>> def funX():
    x = 5
    def funY():
        x *= x
        return x
    return funY

>>> funX()()
Traceback (most recent call last):
  File "<pyshell#47>", line 1, in <module>
    funX()()
  File "<pyshell#46>", line 4, in funY
    x *= x
UnboundLocalError: local variable 'x' referenced before assignment
```

这个报错信息跟之前讲解全局变量的时候基本一样，Python 认为在内部函数的 x 是局部变量的时候，外部函数的 x 就被屏蔽了起来，所以执行 x *= x 的时候，在右边根本就找不到局部变量 x 的值，因此报错。

在 Python3 以前并没有直接的解决方案，只能间接地通过容器类型来存放，因为容器类型不是放在栈里，所以不会被“屏蔽”掉。容器类型这个词儿大家是不是似曾相识？之前介绍的字符串、列表、元组，这些啥都可以往里的放的就是容器类型。于是可以把代码改造如下：

```
>>> def funX():
    x = [5]
    def funY():
        x[0] *= x[0]
        return x[0]
    return funY

>>> funX()()
25
```

到了 Python3 的世界里，有了不少的改进。如果希望在内部函数里可以修改外部函数里的局部变量的值，那么也有一个关键字可以使用，就是 nonlocal，使用方式跟 global 一样：

```
>>> def funX():
    x = 5
    def funY():
        nonlocal x
        x *= x
        return x
    return funY

>>> funX()()
25
```

扩展阅读->游戏中的移动角色：闭包（closure）在实际开发中的作用（地址是 http://bbs.fishc.com/thread-42656-1-1.html）。

## 6.5 lambda 表达式



Python 允许使用 lambda 关键字来创建匿名函数。我们提到一个新的关键字——匿名函数。那什么是匿名函数呢？匿名函数跟普通函数在使用上又有什么不同呢？使用匿名函数又有怎样的优势呢？

那先来谈谈 lambda 表达式怎么用，然后再来讨论它的意义吧。先来定义一个普通的函数：

```
>>> def ds(x):
    return 2 * x + 1

>>> ds(5)
11
```

如果使用 lambda 语句来定义这个函数，就会变成这样：

```
>>> lambda x : 2 * x + 1
<function <lambda> at 0x00000000007FCD08>
```

Python 的 lambda 表达式语法非常精简（符合 Python 的风格），基本语法是在冒号（:）左边放原函数的参数，可以有多个参数，用逗号（,）隔开即可；冒号右边是返回值。在上面的例子中我们发现 lambda 语句实际上是返回一个函数对象，如果要对它进行使用，只需要进行简单的赋值操作即可：

```
>>> g = lambda x : 2 * x + 1
>>> g(5)
11
```

下面演示 lambda 表达式带两个参数的例子：

```
>>> # 这是普通函数:
>>> def add(x, y):
    return x + y

>>> add(3, 4)
7
>>> # 把它转换为 lambda 表达式:
>>> g = lambda x, y : x + y
>>> g(3, 4)
7
```

lambda 表达式的作用：

（1）Python 写一些执行脚本时，使用 lambda 就可以省下定义函数过程，比如说只是需要写个简单的脚本来管理服务器时，就不需要专门定义一个函数然后再写调用，使用 lambda 就可以使得代码更加精简。

（2）对于一些比较抽象并且整个程序执行下来只需要调用一两次的函数，有时候给函数起个名字也是比较头疼的问题，使用 lambda 就不需要考虑命名的问题了。

（3）简化代码的可读性，由于阅读普通函数经常要跳到开头 def 定义的位置，使用 lambda 函数可以省去这样的步骤。

## 介绍两个 BIF：filter()和 map()

接下来给大家介绍 Python 在实际应用中比较实用的两个 BIF，这两个 BIF 使用比起之前一步到位的内建函数来说，相对要复杂一点，也尝试着把今天学到的 lambda 表达式结合在一起。

### 1. filter()

我们研究的第一个内建函数是一个过滤器。我们每天会接触到大量的数据，过滤器的作用就显得非常重要了，通过过滤器，就可以保留你所关注的信息，把其他不感兴趣的东西直接丢掉。这么讲大家应该就了解了，那 Python 的这个 filter()如何来实现过滤的功能呢？先来看下 Python 自己的注释：

```
>>> help(filter)
Help on class filter in module builtins:

class filter(object)
 | filter(function or None, iterable) --> filter object
 |
 | Return an iterator yielding those items of iterable for which function(item)
 | is true. If function is None, return the items that are true.
 …
```

大概意思是：filter 有两个参数。第一个参数可以是一个函数也可以是 None，如果是一个函数的话，则将第二个可迭代数据里的每一个元素作为函数的参数进行计算，把返回 True 的值筛选出来；如果第一个参数为 None，则直接将第二个参数中为 True 的值筛选出来。

这么说有些朋友可能还不大理解，小甲鱼还是用简单的例子帮助解释一下吧：

```
>>> temp = filter(None, [1, 0, False, True])
>>> list(temp)
[1, True]
```

利用 filter()，尝试写一个筛选奇数的过滤器：

```
>>> def odd(x):
	return x % 2

>>> temp = filter(odd, range(10))
>>> list(temp)
[1, 3, 5, 7, 9]
```

那现在学习 lambda 表达式后，完全可以把上述过程转化成一行：

```
>>> list(filter(lambda x : x % 2, range(10)))
[1, 3, 5, 7, 9]
```

### 2. map()

map 在这里不是地图的意思，在编程领域，map 一般作“映射”来解释。map()这个内置函数也有两个参数，仍然是一个函数和一个可迭代序列，将序列的每一个元素作为函数的参数进

行运算加工，直到可迭代序列每个元素都加工完毕，返回所有加工后的元素构成的新序列。

有了刚才 filter()的经验，这里举个例子让大家知道 map()的用法：

```
>>> list(map(lambda x : x * 2, range(10)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

## 6.6 递归



### 6.6.1 递归是“神马”

本节的主题叫递归是“神马”，小甲鱼将通过带感的讲解，来告诉大家“神马”是递归。如果说优秀的程序员是伯乐，那么把递归比喻成神马是再形象不过的了！

递归到底是什么东西呢？有那么厉害吗？为什么大家常说“普通程序员用迭代，天才程序员用递归”呢？没错，通过本节的学习，你将了解递归，通过独立完成课后布置的练习，你将彻底摆脱递归给你生活带来的困扰！

递归这个概念，是算法的范畴，本来不属于 Python 语言的语法内容，但小甲鱼基本在每个编程语言系列教学里都要讲递归，那是因为如果你掌握了递归的方法和技巧，你会发现这是一个非常棒的编程思路！

那么递归算法在日常编程中有哪些例子呢？

汉诺塔游戏如图 6-1 所示。

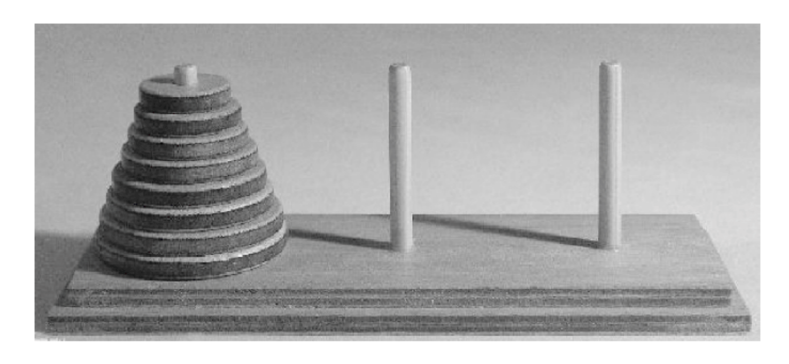


图 6-1　汉诺塔游戏

树结构的定义如图 6-2 所示。

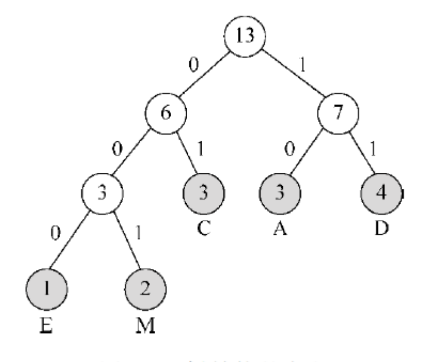



图 6-2　树结构的定义

谢尔宾斯基三角形如图 6-3 所示。

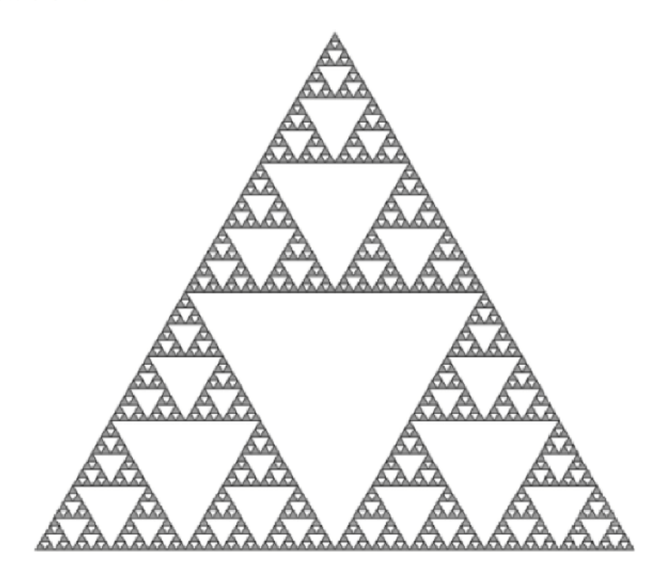


图 6-3 谢尔宾斯基三角形

女神自拍如图 6-4 所示。



图 6-4 女神自拍

说了这么多,在编程上,递归是什么这个概念还没讲呢! 递归,从原理上来说就是函数调用自身这么一个行为。你没听错,在函数内部你可以调用所有可见的函数,当然也包括自己。举个例子:

```
>>> def recursion():
    recursion()

>>> recursion()
Traceback (most recent call last):
```

```
  File "<pyshell#28>", line 1, in <module>
    recursion()
  File "<pyshell#27>", line 2, in recursion
    return recursion()
  File "<pyshell#27>", line 2, in recursion
    return recursion()
  File "<pyshell#27>", line 2, in recursion
return recursion()
…
# 此处省略超多内容
…
RuntimeError: maximum recursion depth exceeded
```

这个例子尝试了初学者玩递归最容易出现的错误。从理论上来讲,这个程序将永远执行下去直至耗尽所有内存资源。不过 Python3 出于“善意的保护”,对递归的深度默认限制是100 层,所以你的代码才会停下来。不过如果你写网络爬虫等工具,可能会爬很深,那你也可以自己设置递归的深度限制。方法如下:

```
>>> import sys
>>> sys.setrecursionlimit(1000000) # 将递归限制设置为 100 万层
```

刚才一来就错误地使用递归把 Python 干掉了,可见递归的厉害和危险,这个后边讲,接下来举个正常的例子跟大家完整解释一下递归。噢,对了,如果你真的设置了一百万层递归,那么一不小心又玩脱了,Python 可能会卡在那里很久,这时你可以通过 Ctrl+C 让它停止哦。

### 6.6.2 写一个求阶乘的函数

正整数的阶乘是指从 1 乘以 2 乘以 3 乘以 4 一直乘到所要求的数。例如所要求的数是 5,则阶乘式是 1×2×3×4×5,得到的积是 120,所以 120 就是 5 的阶乘。好,那大家先自己尝试下实现一个非递归版本:

```
# p6_5.py
def recursion(n):
    result = n
    for i in range(1, n):
        result *= i
    return result

number = int(input('请输入一个整数: '))
result = recursion(number)
print("%d 的阶乘是: %d" % (number, result))
```

程序实现结果如下:

```
>>>
请输入一个正整数: 5
5 的阶乘是: 120
>>>
```

普通函数的实现相信大家都会写,那再来演示一下递归版本:

```
# p6_6.py
```

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

number = int(input('请输入一个整数: '))
result = factorial(number)
print("%d 的阶乘是: %d" % (number, result))
```

以前没接触过递归的小伙伴肯定会怀疑这是否能正常执行？没错，这完全符合递归的预期和标准，所以函数无疑可以正确执行并返回正确的结果！程序实现结果跟上边的结果是一样的：

```
>>>
请输入一个正整数：5
5 的阶乘是：120
>>>
```

我们说过，麻雀虽小，却五脏俱全。这个例子满足了递归的两个条件：

(1) 调用函数本身。

(2) 设置了正确的返回条件。

上面程序的详细分析如图 6-5 所示。

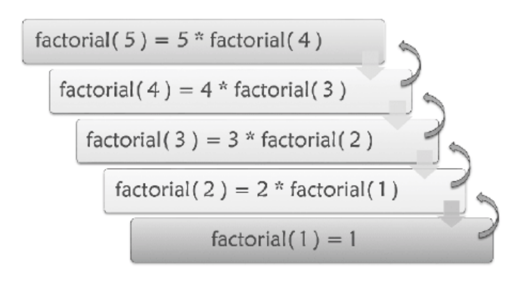



图 6-5 递归函数的实现分析

最后要郑重说一下“普通程序员用迭代，天才程序员用递归”这句话是不无道理的。但是你不要理解错了，不是说会使用递归，把所有能迭代的东西用递归来代替就是“天才程序员”了，恰好相反，如果你真的这么做的话，那你就是“乌龟程序员”啦。为什么这么说呢？不要忘了，递归的实现可以是函数自个儿调用自个儿，每次函数的调用都需要进行压栈、弹栈、保存和恢复寄存器的栈操作，所以在这上边是非常消耗时间和空间的。

另外，如果递归一旦忘记了返回，或者错误地设置了返回条件，那么执行这样的递归代码就会变成一个无底洞：只进不出！所以在写递归代码的时候，千万要记住口诀：递归递归，归去来兮！

因此，结合以上两点致命缺陷，很多初学者经常就会在论坛上讨论递归存在的必要性，他们认为递归完全没必要，用循环就可以实现。其实这就跟讨论 C 语言好还是 Python 优秀一样，是没有必要的。因为一样东西既然能够持续存在，那必然有它存在的道理。递归用在妙处，自然代码简洁、精练，所以说“天才程序员使用递归”。

### 6.6.3 这帮小兔崽子

按理来说，今天的话题跟兔子不拉边，不过嘛，大家也知道小甲鱼的风格了，天南地北，总有相关联的东西可以拉过来扯淡。本节就用递归来实现斐波那契(Fibonacci)数列吧。

斐波那契数列的发明者，是意大利数学家列昂纳多·斐波那契(Leonardo Fibonacci)。这老头说来跟小甲鱼也有一定的渊源，就是老爱拿动物交配说事儿，不同的是小甲鱼注重过程和细节，而这老头更关心结果，下边就有一个他讲过的故事：如果说兔子在出生两个月后，就有繁殖能力，在拥有繁殖能力之后，这对兔子每个月能生出一对小兔子来。假设所有兔子都不会死去，能够一直干下去，那么一年之后可以繁殖多少对兔子呢？

我们都知道兔子繁殖能力是惊人的，如图6-6所示。

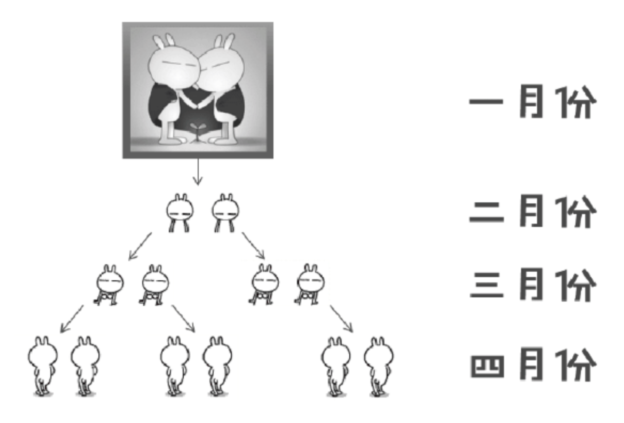



图6-6　斐波那契数列

数据统计如表6-1所示。

**表6-1　斐波那契数列**

| 所经过的月数 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 兔子的总对数 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 |

可以用数学函数来定义，如图6-7所示。

$$F(n)=\begin{cases}1, & \text{当 } n=1 \text{ 时}\\ 1, & \text{当 } n=2 \text{ 时}\\ F(n-1)+F(n-2), & \text{当 } n>2 \text{ 时}\end{cases}$$

图6-7　求斐波那契数列的公式

假设需要求出经历了20个月后，总共有多少对小兔崽子，不妨一起考虑一下分别用迭代和递归如何实现？

迭代实现：

```
# p6_7.py
def fab(n):
    a1 = 1
    a2 = 1
    a3 = 1
```

```
    if n < 1:
        print('输入有误!')
        return -1
    while (n-2) > 0:
        a3 = a1 + a2
        a1 = a2
        a2 = a3
        n -= 1
    return a3

result = fab(20)
if result != -1:
    print('总共有%d对小兔崽子诞生!' % result)
```

接下来看看递归的实现原理，如图6-8所示。

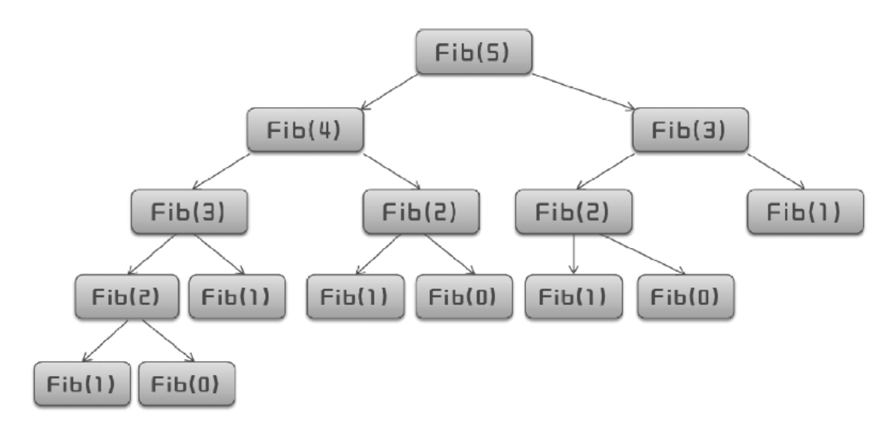



图6-8 递归实现斐波那契数列的原理

递归实现：

```
# p6_8.py
def fab(n):
    if n < 1:
        print('输入有误!')
        return -1
    if n == 1 or n == 2:
        return 1
    else:
        return fab(n-1) + fab(n-2)

result = fab(20)
if result != -1:
print('总共有%d对小兔崽子诞生!' % result)
```

可见逻辑非常简单，直接把想的东西写成代码就是递归算法了。不过，之前我们总说递归如果使用不当，效率会很低，但是有多低呢？我们这就来证明一下。我们试图把20个月修改为35个月，然后试试看把程序执行起来……

发现了吧，用迭代代码来实现基本是毫秒级的，而用递归来实现就考验你的CPU能力啦（*N*秒～*N*分钟不等）。这就是小甲鱼不支持大家所有东西都用递归求解的原因，本来好好的

一个代码,给你用了递归,效率反而拉下了一大截。

为了体现递归正确使用的优势,下一节我们来谈谈利用递归解决汉诺塔难题。如果你不懂得递归,试图想要写个程序来解决问题是相当困难的,但如果使用了递归,你会发现问题奇迹般的变简单了!

这里可以在线玩这个游戏,大家不妨边玩边思考代码该怎么实现:http://www.kaixin001.com/flashgame/game/10406.html。

## 6.6.4 汉诺塔

看过小甲鱼其他教程的"鱼油"们可能会说,怎么一谈到递归算法就要拿汉诺塔来举例呢?没办法,因为小甲鱼小时候太笨了,这个游戏老是玩不过关,好不容易在自学编程的时候,也卡在这里好长一段时间,所以呢,现在懂了啊,可以得瑟了嘛!

汉诺塔(如图6-9所示)的来源据说是这样的:一位法国数学家曾编写过一个印度的古老传说:说的是,在世界中心贝拿勒斯的圣庙里边,有一块黄铜板,上边插着三根宝针。印度教的主神梵天在创造世界的时候,在其中一根针上从下到上地穿好了由大到小的64片金片,这就是所谓的汉诺塔。然后不论白天或者黑夜,总有一个僧侣在按照下面的法则来移动这些金片:"一次只移动一片,不管在哪根针上,小片必须在大片上面。"规则很简单,另外僧侣们预言,当所有的金片都从梵天穿好的那根针上移到另外一根针上时,世界就将在一声霹雳中消灭,而梵塔、庙宇和众生也都将同归于尽。

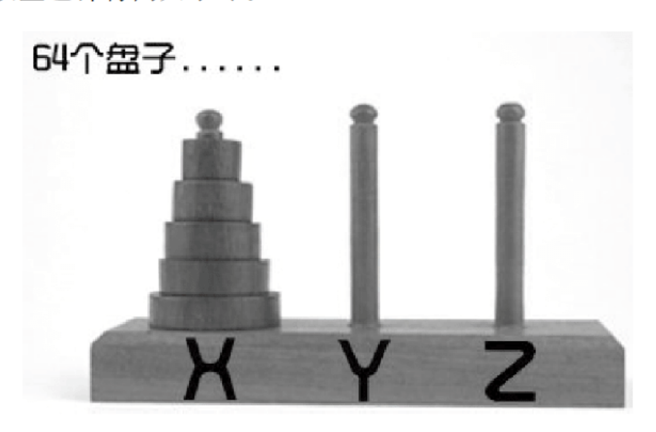



图6-9 汉诺塔

要解决一个问题,大家说什么最重要?没错,思路!思路有了,问题就可以随之迎刃而解。游戏上节课我们也玩了,小甲鱼可不是让大家白玩的,玩过之后大家心里有底,现在的分析大家才容易听进去。

对于游戏的玩法,可以简单分解为三个步骤:

(1)将前63个盘子从X移动到Y上,确保大盘在小盘下。

(2)将最底下的第64个盘子从X移动到Z上。

(3)将Y上的63个盘子移动到Z上。

这样看上去问题就简单一点了,有些鱼油说小甲鱼你这不废话嘛!?有说跟没说一样!因为关键在于步骤(1)和步骤(3)应该如何执行才是让人头疼的问题。

但是你仔细思考下,在游戏中,我们发现由于每次只能移动一个圆盘,所以在移动的过程

中显然要借助另外一根针才可以实施。也就是说，步骤(1)将1～63个盘子需要借助Z移到Y上，步骤(3)将Y针上的63个盘子需要借助X移到Z针上。

所以把新的思路聚集为以下两个问题：

问题一，将X上的63个盘子借助Z移到Y上；

问题二，将Y上的63个盘子借助X移到Z上。

然后我们惊奇地发现，解决这两个问题的方法跟刚才第一个问题的思路是一样的，都可以拆解成三个步骤来实现。

问题一(将X上的63个盘子借助Z移到Y上)拆解为：

(1) 将前62个盘子从X移动到Z上，确保大盘在小盘下。

(2) 将最底下的第63个盘子移动到Y上。

(3) 将Z上的62个盘子移动到Y上。

问题二(将Y上的63个盘子借助X移到Z上)拆解为：

(1) 将前62个盘子从Y移动到X上，确保大盘在小盘下。

(2) 将最底下的第63个盘子移动到Z上。

(3) 将X上的62个盘子移动到Y上。

说到这里，是不是发现了什么？没错，汉诺塔的拆解过程刚好满足递归算法的定义，因此，对于如此难题，使用递归来解决，问题就变得相当的简单。参考代码：

```
# p6_9.py
def hanoi(n, x, y, z):
    if n == 1:
        print(x, ' --> ', z) # 如果只有一层,直接从 x 移动到 z
    else:
        hanoi(n-1, x, z, y) # 将前 n-1 个盘子从 X 移动到 Y 上
        print(x, ' --> ', z) # 将最底下的第 64 个盘子从 X 移动到 Z 上
        hanoi(n-1, y, x, z) # 将 Y 上的 63 个盘子移动到 Z 上

n = int(input('请输入汉诺塔的层数: '))
hanoi(n, 'X', 'Y', 'Z')
```

看，这就是递归的魔力！

# 第7章 字典和集合

## 7.1 字典：当索引不好用时



有天你想翻开《新华字典》，查找下“龟”是不是一种鸟。如果是按拼音检索，你总不可能从字母 a 开始查找吧？你应该直接翻到字母 g 在字典中的位置，然后接着找到 gui 的发音，继而找到“龟”字的释义：广义上指龟鳖目的统称，狭义上指龟科下的物种。

在 Python 中也有字典，就拿刚才的例子来说，Python 的字典把这个字（或单词）称为“键(key)”，把其对应的含义称为“值(value)”。另外值得提一下的是，Python 的字典在有些地方称为哈希(hash)，有些地方称为关系数组，其实这些都跟今天要讲的 Python 字典是同一个概念。

字典是 Python 中唯一的映射类型，映射是数学上的一个术语，指两个元素集之间元素相互“对应”的关系，如图 7-1 所示。

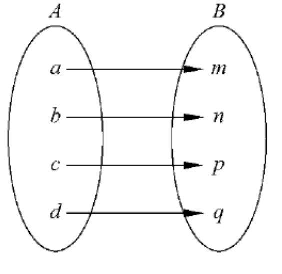



图 7-1 映射

映射类型区别于序列类型，序列类型以数组的形式存储，通过索引的方式来获取相应位置的值，一般索引值与对应位置存储的数据是毫无关系的。举个例子：

```
>>> brand = ["李宁", "耐克", "阿迪达斯", "鱼C工作室"]
>>> slogan = ["一切皆有可能","Just do it","Impossible is nothing","让编程改变世界"]
>>> print("鱼C工作室的口号是: ", slogan[brand.index("鱼C工作室")])
鱼C工作室的口号是: 让编程改变世界
```

列表 brand、slogan 的索引和相对的值是没有任何关系的，可以看出，唯一有联系的就是两个列表间，索引号相同的元素是有关系的（品牌对应口号嘛），所以这里通过“brand.index('鱼C工作室')”这样的语句，间接地实现通过品牌查找对应的口号的功能。

这确实是一种可实现方法，但用起来多少有些别扭，而且效率还不高。况且 Python 是以简洁为主，这样的实现肯定是差强人意的。所以，需要有字典这种映射类型的出现。

### 7.1.1 创建和访问字典

先演示一下用法：

```
>>> dict1 = {"李宁":"一切皆有可能", "耐克":"Just do it", "阿迪达斯":"Impossible is nothing",
"鱼C工作室":"让编程改变世界"}
```

```
>>> dict1
{'李宁': '一切皆有可能', '阿迪达斯': 'Impossible is nothing', '鱼C工作室': '让编程改变世界', '耐克': 'Just do it'}
>>> print("鱼C工作室的口号是:", dict1['鱼C工作室'])
```

字典的使用非常简单,它有自己的标志性符号,就是用大括号({})定义。字典由多个键及其对应的值共同构成,每一对键值组合称为项。在刚才的例子中,“李宁”、“耐克”、“阿迪达斯”、“鱼C工作室”这些品牌就是键,而“一切皆有可能”、“Just do it”、“Impossible is nothing”、“让编程改变世界”这些口号就是对应的值。眼尖的读者可能已经发现了:字典中的项跟创建的顺序是不一样的? 没错,字典跟序列不同,序列讲究顺序,字典讲究映射,不讲顺序。

另外需要注意的是:字典的键必须独一无二,而值可以取任何数据类型,但必须是不可变的(如字符串、数或元组)。

要声明一个空字典,直接用个大括号即可:

```
>>> empty = {}
>>> empty
{}
>>> type(empty)
<class 'dict'>
```

你也可以用dict()来创建字典:

```
>>> dict1 = dict((('F', 70), ('i', 105), ('s', 115), ('h', 104), ('C', 67)))
>>> dict1
{'s': 115, 'C': 67, 'F': 70, 'h': 104, 'i': 105}
```

有读者朋友可能会问,为什么上面的例子中这么多括号?

因为dict()函数的参数可以是一个序列(但不能是多个),所以要打包成一个元组序列(列表也可以)。当然,如果嫌上面的做法太麻烦,还可以通过提供具有映射关系的参数来创建字典:

```
>>> dict1 = dict(F=70, i=105, s=115, h=104, C=67)
>>> dict1
{'C': 67, 's': 115, 'F': 70, 'h': 104, 'i': 105}
```

这里要注意的是键的位置不能加上字符串的引号,否则会报错:

```
>>> dict1 = dict('F'=70, 'i'=105, 's'=115, 'h'=104, 'C'=67)
SyntaxError: keyword can't be an expression
```

还有一种创建方法是直接给字典的键赋值,如果键存在,则改写键对应的值;如果不存在,则创建一个新的键并赋值:

```
>>> dict1
{'C': 67, 's': 115, 'F': 70, 'h': 104, 'i': 105}
>>> dict1['x'] = 88
>>> dict1
{'s': 115, 'h': 104, 'i': 105, 'C': 67, 'x': 88, 'F': 70}
>>> dict1['x'] = 120
>>> dict1
{'s': 115, 'h': 104, 'i': 105, 'C': 67, 'x': 120, 'F': 70}
```

正所谓殊途同归,下面列举的五种方法都是创建同样的字典,请大家仔细体会下:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one':1, 'two':2, 'three':3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three':3, 'one':1, 'two':2})
>>> a == b == c == d == e
True
```

## 7.1.2 各种内置方法

字典是 Python 中唯一的映射类型，字典不是序列。如果在序列中试图为一个不存在的位置赋值的时候，会报错；但是如果是在字典中，会自动创建相应的键并添加对应的值进去。

### 1. fromkeys()

fromkeys()方法用于创建并返回一个新的字典，它有两个参数：第一个参数是字典的键；第二个参数是可选的，是传入键对应的值。如果不提供，那么默认是 None。举个例子：

```
>>> dict1 = {}
>>> dict1.fromkeys((1, 2, 3))
{1: None, 2: None, 3: None}
>>> dict2 = {}
>>> dict2.fromkeys((1, 2, 3), "Number")
{1: 'Number', 2: 'Number', 3: 'Number'}
>>> dict3 = {}
>>> dict3.fromkeys((1, 2, 3), ("one", "two", "three"))
{1: ('one', 'two', 'three'), 2: ('one', 'two', 'three'), 3: ('one', 'two', 'three')}
```

上面最后一个例子告诉我们做事不能总是想当然，有时候现实会给你狠狠的一棒。fromkeys()方法并不会将值"one"、"two"和"three"分别赋值键 1、2 和 3，因为 fromkeys()把("one", "two", "three")当成一个值了。

### 2. keys()、values()和 items()

访问字典的方法有 keys()、values()和 items()。

keys()用于返回字典中的键，values()用于返回字典中所有的值，那么 items()当然就是返回字典中所有的键值对（也就是项）啦。举个例子：

```
>>> dict1 = {}
>>> dict1 = dict1.fromkeys(range(32), "赞")
>>> dict1.keys()
dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31])
>>> dict1.values()
dict_values(['赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞',
'赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞', '赞'])
>>> dict1.items()
dict_items([(0, '赞'), (1, '赞'), (2, '赞'), (3, '赞'), (4, '赞'), (5, '赞'), (6, '赞'), (7, '赞'), (8,
'赞'), (9, '赞'), (10, '赞'), (11, '赞'), (12, '赞'), (13, '赞'), (14, '赞'), (15, '赞'), (16, '赞'),
(17, '赞'), (18, '赞'), (19, '赞'), (20, '赞'), (21, '赞'), (22, '赞'), (23, '赞'), (24, '赞'), (25, '赞
```

```
'), (26, '赞'), (27, '赞'), (28, '赞'), (29, '赞'), (30, '赞'), (31, '赞')])
```

字典可以很大，有些时候我们并不知道提供的项是否在字典中存在，如果不存在，Python就会报错：

```
>>> print(dict1[32])
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    print(dict1[32])
KeyError: 32
```

对于代码调试阶段，报错让程序员及时发现程序存在的问题并修改之。但是如果程序面向用户了，那么经常报错的程序肯定会被用户所遗弃……

### 3. get()

get()方法提供了更宽松的方式去访问字典项，当键不存在的时候，get()方法并不会报错，只是默默地返回了一个 None，表示啥都没找到：

```
>>> dict1.get(31)
'赞'
>>> dict1.get(32)
>>>
```

如果希望找不到数据时返回指定的值，那么可以在第二个参数设置对应的默认返回值：

```
>>> dict1.get(32, "木有")
'木有'
```

如果不知道一个键是否在字典中，那么可以使用成员资格操作符(in 或 not in)来判断：

```
>>> 31 in dict1
True
>>> 32 in dict2
False
```

在字典中检查键的成员资格比序列更高效，当数据规模相当大的时候，两者的差距会很明显(注：因为字典是采用哈希的方法一对一找到成员，而序列则是采取迭代的方式逐个比对)。最后要注意的一点是，这里查找的是键而不是值，但是在序列中查找的是元素的值而不是元素的索引。

如果需要清空一个字典，则使用 clear()方法：

```
>>> dict1
{0: '赞', 1: '赞', 2: '赞', 3: '赞', 4: '赞', 5: '赞', 6: '赞', 7: '赞', 8: '赞', 9: '赞', 10: '赞', 11:
'赞', 12: '赞', 13: '赞', 14: '赞', 15: '赞', 16: '赞', 17: '赞', 18: '赞', 19: '赞', 20: '赞', 21: '赞',
22: '赞', 23: '赞', 24: '赞', 25: '赞', 26: '赞', 27: '赞', 28: '赞', 29: '赞', 30: '赞', 31: '赞'}
>>> dict1.clear()
>>> dict1
{}
```

有的读者可能会使用变量名赋值为一个空字典的方法来清空字典，这样做存在一定的弊端。举个例子跟大家说说两种清除方法有什么不同：

```
>>> a = {"姓名":"小甲鱼"}
>>> b = a
>>> b
{'姓名': '小甲鱼'}
>>> a = {}
>>> a
{}
>>> b
{'姓名': '小甲鱼'}
```

从上面的例子中可以看到，a、b指向同一个字典，然后试图通过将a重新指向一个空字典来达到清空的效果时，我们发现原来的字典并没有被真正清空，只是a指向了一个新的空字典而已。所以这种做法在一定条件下会留下安全隐患（例如，账户的数据和密码等资料有可能会被窃取）。

推荐的做法是使用clear()方法：

```
>>> a = {"姓名":"小甲鱼"}
>>> b = a
>>> b
{'姓名': '小甲鱼'}
>>> a.clear()
>>> a
{}
>>> b
{}
```

#### 4. copy()

copy()方法是复制字典：

```
>>> a = {1:"one", 2:"two", 3:"three"}
>>> b = a.copy()
>>> id(a)
63239624
>>> id(b)
63239688
>>> a[1] = "four"
>>> a
{1: 'four', 2: 'two', 3: 'three'}
>>> b
{1: 'one', 2: 'two', 3: 'three'}
```

#### 5. pop()和popitem()

接下来讲讲pop()和popitem()，pop()是给定键弹出对应的值，而popitem()是弹出一个项，这两个比较容易：

```
>>> a = {1:"one", 2:"two", 3:"three", 4:"four"}
>>> a.pop(2)
'two'
>>> a
{1: 'one', 3: 'three', 4: 'four'}
>>> a.popitem()
```

```
(1, 'one')
>>> a
{3: 'three', 4: 'four'}
```

setdefault()方法和get()方法有点相似，但是setdefault()在字典中找不到相应的键时会自动添加：

```
>>> a = {1:"one", 2:"two", 3:"three", 4:"four"}
>>> a.setdefault(3)
'three'
>>> a.setdefault(5)
>>> a
{1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: None}
```

### 6. update()

最后一个是update()方法，可以利用它来更新字典：

```
>>> pets = {"米奇":"老鼠", "汤姆":"猫", "小白":"猪"}
>>> pets.update(小白 = "狗")
>>> pets
{'米奇': '老鼠', '汤姆': '猫', '小白': '狗'}
```

还记得在6.2节的末尾我们埋下了一个伏笔，在末尾讲到收集参数的时候，我们说Python还有另一种收集方式，就是用两个星号(**)表示。两个星号的收集参数表示为将参数们打包成字典的形式，现在讲到了字典，就顺理成章地给大家讲讲吧。

收集参数其实有两种打包形式：一种是以元组的形式打包，另一种则是以字典的形式打包：

```
>>> def test(**params):
    print("有 %d 个参数" % len(params))
    print("它们分别是：", params)

>>> test(a=1, b=2, c=3, d=4, e=5)
有 5 个参数
它们分别是：{'d': 4, 'e': 5, 'b': 2, 'c': 3, 'a': 1}
```

当参数带两个星号(**)时，传递给函数的任意个key=value实参会被打包进一个字典中。那么有打包就有解包，来看一个例子：

```
>>> a = {"one":1, "two":2, "three":3}
>>> test(**a)
有 3 个参数
它们分别是：{'three': 3, 'one': 1, 'two': 2}
```

## 7.2 集合：在我的世界里，你就是唯一



上节讲了Python中的字典，Python的字典是对数学中映射概念支持的直接体现。而今天呢，我们请来了字典的表亲：集合。

喔？难道它们长得很像？来，大家看下代码：

```
>>> num1 = {}
>>> type(num1)
<class 'dict'>
>>> num2 = {1, 2, 3, 4, 5}
>>> type(num2)
<class 'set'>
```

你们确实没有眼花，在Python3里，如果用大括号括起一堆数字但没有体现映射关系，那么Python就会认为这堆玩意儿就是个集合。

那集合有什么特色呢？不知道大家有没有注意到本节的标题——"集合：在我的世界里，你就是唯一"？

好，说回主题，集合在Python中几乎起到的所有作用就是两个字：唯一。举个例子：

```
>>> num = {1, 2, 3,4 ,5, 4, 3, 2, 1}
>>> num
{1, 2, 3, 4, 5}
```

大家看到，我们根本不需要做什么，集合就会帮我们把重复的数据清理掉，这样是不是很方便呢？但要注意的是，集合是无序的，也就是你不能试图去索引集合中的某一个元素：

```
>>> num[2]
Traceback (most recent call last):
  File "<pyshell#81>", line 1, in <module>
    num[2]
TypeError: 'set' object does not support indexing
```

### 7.2.1 创建集合

创建集合有两种方法：一种是直接把一堆元素用大括号({})括起来；另一种是用set()。

```
>>> set1 = {"小甲鱼", "小鱿鱼", "小护士", "小甲鱼"}
>>> set2 = set(["小甲鱼", "小鱿鱼", "小护士", "小甲鱼"])
>>> set1 == set2
True
```

现在要求去除列表[1，2，3，4，5，5，3，1，0]中重复的元素。如果还没有学习过集合，你可能会这么写：

```
>>> list1 = [1, 2, 3, 4, 5, 5, 3, 1, 0]
>>> temp = list1[:]
>>> list1.clear()
>>> for each in temp:
        if each not in list1:
            list1.append(each)

>>> list1
[1, 2, 3, 4, 5, 0]
```

当你学习了集合之后，就可以这么干：

```
>>> list1 = [1, 2, 3, 4, 5, 5, 3, 1, 0]
>>> list1 = list(set(list1))
>>> list1
[0, 1, 2, 3, 4, 5]
```

看，知识才是第一生产力！不过大家发现没有？由于 set()创造了的集合内部是无序的，所以再调用 list()将无序的集合转换成列表就不能保证原来的列表的顺序了（这里 Python 好心办坏事儿，把 0 放到前边了），所以如果关注列表中元素的前后顺序问题，使用 set()这个函数时就要提高警惕啦！

## 7.2.2 访问集合

由于集合中的元素是无序的，所以并不能像序列那样用下标来进行访问，但是可以使用迭代把集合中的数据一个个读取出来：

```
>>> set1 = {1, 2, 3, 4, 5, 4, 3, 2, 1, 0}
>>> for each in set1:
     print(each, end = ' ')

0 1 2 3 4 5
```

当然也可以使用 in 和 not in 判断一个元素是否在集合中已经存在：

```
>>> 0 in set1
True
>>> 'oo' in set1
False
>>> 'xx' not in set1
True
```

使用 add()方法可以为集合添加元素，使用 remove()方法可以删除集合中已知的元素：

```
>>> set1.add(6)
>>> set1
{0, 1, 2, 3, 4, 5, 6}
>>> set1.remove(5)
>>> set1
{0, 1, 2, 3, 4, 6}
```

## 7.2.3 不可变集合

有些时候希望集合中的数据具有稳定性，也就是说，像元组一样不能随意地增加或删除集合中的元素。那么我们可以定义不可变集合，这里使用的是 frozenset()函数，没错，就是把元素给 frozen（冰冻）起来：

```
>>> set1 = frozenset({1, 2, 3, 4, 5})
>>> set1.add(6)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in <module>
    set1.add(6)
AttributeError: 'frozenset' object has no attribute 'add'
```




# 永久存储

## 8.1 文件：因为懂你，所以永恒



大多数的程序都遵循着：输入->处理->输出的模型，首先接收输入数据，然后按照要求进行处理，最后输出数据。到目前为止，我们已经很好地了解了如何处理数据，然后打印出需要的结果。不过你可能已经胃口大开，不再只满足于使用 input 接收用户输入，使用 print 输出处理结果了。你迫切想要的是关注到系统的方方面面，你需要自己的代码可以自动分析系统的日志，你需要分析的结果可以保存为一个新的日志，甚至你需要跟外面的世界进行交流。

相信大家都曾经有这样的经历：当你在编写代码写起劲儿的时候，系统突然蓝屏崩溃了，重启之后发现刚才写入的代码都不见了，这时候你就会吐槽这破系统怎么这么不稳定等等。

在你编写代码的时候，操作系统为了更快地做出响应，把所有当前的数据都放在内存中，因为内存和 CPU 数据传输的速度要比在硬盘和 CPU 之间传输的速度快很多倍。但内存有一个天生的不足，就是一旦断电就没戏，所以小甲鱼在这里再一次呼吁广大未来即将成为伟大程序员的读者们：请养成一个优雅的习惯，随时使用快捷键 Ctrl+S 保存你的数据。

由于 Windows 是以扩展名来指出文件是什么类型，所以相信很多习惯使用 Windows 的朋友很快就反应过来了，.exe 是可执行文件格式，.txt 是文本文件，.ppt 是 PowerPoint 的专用格式……所有这些都称为文件。

### 8.1.1 打开文件

在 Python 中，使用 open()这个函数来打开文件并返回文件对象：

```
open(file, mode = 'r', buffering = - 1, encoding = None, errors = None, newline = None, closefd =
True, opener = None)
```

open()这个函数有很多参数，但作为初学者的我们，只需要先关注第一个和第二个参数即可。第一个参数是传入的文件名，如果只有文件名，不带路径的话，那么 Python 会在当前文件夹中去找到该文件并打开。有的读者可能会问：如果我要打开的文件事实上并不存在呢？

那就要看第二个参数了，第二个参数指定文件打开模式，如表 8-1 所示。

表 8-1 文件的打开模式

| 打开模式 | 执 行 操 作 |
| --- | --- |
| 'r' | 以只读方式打开文件(默认) |
| 'w' | 以写入的方式打开文件,会覆盖已存在的文件 |
| 'x' | 如果文件已经存在,使用此模式打开将引发异常 |
| 'a' | 以写入模式打开,如果文件存在,则在末尾追加写入 |
| 'b' | 以二进制模式打开文件 |
| 't' | 以文本模式打开(默认) |
| '+' | 可读写模式(可添加到其他模式中使用) |
| 'U' | 通用换行符支持 |

使用 open()成功打开一个文件之后,它会返回一个文件对象,拿到这个文件对象,就可以读取或修改这个文件:

```
>>> # 先将 record.txt 文件放到 Python 的根目录下(如 C:\Python34)
>>> f = open("record.txt")
```

没有消息就是好消息,说明我们的文件成功被打开了。

### 8.1.2 文件对象的方法

打开文件并取得文件对象之后,就可以利用文件对象的一些方法对文件进行读取或修改等操作。表 8-2 列举了平时常用的一些文件对象方法。

表 8-2 文件对象方法

| 文件对象的方法 | 执 行 操 作 |
| --- | --- |
| close() | 关闭文件 |
| read(size=-1) | 从文件读取 size 个字符,当未给定 size 或给定负值的时候,读取剩余的所有字符,然后作为字符串返回 |
| readline() | 从文件中读取一整行字符串 |
| write(str) | 将字符串 str 写入文件 |
| writelines(seq) | 向文件写入字符串序列 seq,seq 应该是一个返回字符串的可迭代对象 |
| seek(offset, from) | 在文件中移动文件指针,从 from(0 代表文件起始位置,1 代表当前位置,2 代表文件末尾)偏移 offset 个字节 |
| tell() | 返回当前在文件中的位置 |

### 8.1.3 文件的关闭

close()方法用于关闭文件。如果是讲 C 语言编程教学,小甲鱼一定会一万次地强调文件的关闭非常重要。而 Python 拥有垃圾收集机制,会在文件对象的引用计数降至零的时候自动关闭文件,所以在 Python 编程里,如果忘记关闭文件并不会造成内存泄露那么危险的结果。

但并不是说就可以不要关闭文件,如果你对文件进行了写入操作,那么应该在完成写入之后关闭文件。因为 Python 可能会缓存你写入的数据,如果中途发生类似断电之类的事故,那些缓存的数据根本就不会写入到文件中。所以,为了安全起见,要养成使用完文件后立刻关闭

的好习惯。

### 8.1.4 文件的读取和定位

文件的读取方法很多，可以使用文件对象的 read()和 readline()方法，也可以直接 list(f)或者直接使用迭代来读取。read()是按字节为单位读取，如果不设置参数，那么会全部读取出来，文件指针指向文件末尾。tell()方法可以告诉你当前文件指针的位置：

```
>>> f.read()
'小客服:小甲鱼,有个好评很好笑哈.\n 小甲鱼:哦?\n 小客服:"有了小甲鱼,以后妈妈再也不用担心我的学习了～"\n 小甲鱼:哈哈哈,我看到吖,我还发微博了呢～\n 小客服:嗯嗯,我看了你的微博吖～\n 小甲鱼:OK～\n 小客服:那个有条回复"左手拿着小甲鱼,右手拿着打火机,哪里不会点哪里,so easy ^_^"\n 小甲鱼:T_T'
>>> f.tell()
284
```

刚才提到的文件指针是啥？你可以认为它是一个“书签”，起到定位的作用。使用 seek()方法可以调整文件指针的位置。seek(offset, from)方法有两个参数，表示从 from(0 代表文件起始位置，1 代表当前位置，2 代表文件末尾)偏移 offset 字节。因此将文件指针设置到文件起始位置，使用 seek(0, 0)即可：

```
>>> f.tell()
284
>>> f.seek(0, 0)
0
>>> f.read(5)
'小客服:小'
>>> f.tell()
9
```

(注：因为 1 个中文字符占用 2 个字节的空间，所以 4 个中文加 1 个英文冒号刚好到位置 9。)

readline()方法用于在文件中读取一整行，就是从文件指针的位置向后读取，直到遇到换行符(\n)结束：

```
>>> f.readline()
'甲鱼,有个好评很好笑哈.\n'
```

此前介绍过列表的强大，说什么都可以往里放，这不，也可以把整个文件的内容放到列表中：

```
>>> list(f)
['小甲鱼:哦?\n', '小客服:"有了小甲鱼,以后妈妈再也不用担心我的学习了～"\n', '小甲鱼:哈哈哈,我看到吖,我还发微博了呢～\n', '小客服:嗯嗯,我看了你的微博吖～\n', '小甲鱼:OK～\n', '小客服:那个有条回复"左手拿着小甲鱼,右手拿着打火机,哪里不会点哪里,so easy ^_^"\n', '小甲鱼:T_T']
```

对于迭代读取文本文件中的每一行，有些读者可能会这么写：

```
>>> f.seek(0, 0)
0
>>> lines = list(f)
```

```
>>> for each_line in lines:
    print(each_line)
```

这样写并没有错，但给人的感觉就像是你拿酒精灯去烧开水，水是烧得开，不过效率不是很高。因为文件对象自身是支持迭代的，所以没必要绕圈子，直接使用 for 语句把内容迭代读取出来即可：

```
>>> f.seek(0, 0)
0
>>> for each_line in f:
    print(each_line)
```

## 8.1.5 文件的写入

如果需要写入文件，请确保之前的打开模式有'w'或'a'，否则会出错：

```
>>> f = open("record.txt")
>>> f.write("这是一段待写入的数据")
Traceback (most recent call last):
  File "<pyshell#135>", line 1, in <module>
    f.write("这是一段待写入的数据")
io.UnsupportedOperation: not writable
>>> f.close()
>>> f = open("record.txt", "w")
>>> f.write("这是一段待写入的数据")
10
>>> f.close()
```

然而一定要小心的是：使用'w'模式写入文件，此前的文件内容会被全部删除！如图 8-1 所示，小甲鱼和小客服的对话备份已经不在了。




图 8-1 'w'打开模式会删除原来的文件内容

如果要在原来的内容上追加，一定要使用'a'模式打开文件哦。这是血淋淋的教训，不要问我为什么(想想都是泪啊)！

## 8.1.6 一个任务



本节要求读者朋友独立来完成一个任务——将文件(record2.txt)中的数据进行分割并按照以下规则保存起来：

(1) 将小甲鱼的对话单独保存为 boy_＊.txt 的文件(去掉“小甲鱼:”)。

(2) 将小客服的对话单独保存为 girl_＊.txt 的文件(去掉“小客服:”)。

(3) 文件中总共有三段对话,分别保存为 boy_1.txt、girl_1.txt、boy_2.txt、girl_2.txt、boy_3.txt、girl_3.txt 共 6 个文件(提示:文件中不同的对话间已经使用“==========================================”分割)。

大家一定要自己先动动手再参考答案哦:

```
# p8_1.py
count = 1
boy = []
girl = []
f = open('record.txt')
for each_line in f:
    if each_line[:6] != '======':
        (role, line_spoken) = each_line.split(':', 1)
        if role == '小甲鱼':
            boy.append(line_spoken)
        if role == '小客服':
            girl.append(line_spoken)
    else:
        file_name_boy = 'boy_' + str(count) + '.txt'
        file_name_girl = 'girl_' + str(count) + '.txt'
        boy_file = open(file_name_boy, 'w')
        girl_file = open(file_name_girl, 'w')
        boy_file.writelines(boy)
        girl_file.writelines(girl)
        boy = []
        girl = []
        count += 1
file_name_boy = 'boy_' + str(count) + '.txt'
file_name_girl = 'girl_' + str(count) + '.txt'
boy_file = open(file_name_boy, 'w')
girl_file = open(file_name_girl, 'w')
boy_file.writelines(boy)
girl_file.writelines(girl)
boy_file.close()
girl_file.close()
f.close()
```

事实上可以利用函数封装得更好看一些:

```
# p8_2.py
def save_file(boy, girl, count):
    file_name_boy = 'boy_' + str(count) + '.txt'
    file_name_girl = 'girl_' + str(count) + '.txt'
    boy_file = open(file_name_boy, 'w')
    girl_file = open(file_name_girl, 'w')
    boy_file.writelines(boy)
    girl_file.writelines(girl)
    boy_file.close()
    girl_file.close()
```

```
def split_file(file_name):
    count = 1
    boy = []
    girl = []
    f = open(file_name)
    for each_line in f:
        if each_line[:6] != '======':
            (role, line_spoken) = each_line.split(':', 1)
            if role == '小甲鱼':
                boy.append(line_spoken)
            if role == '小客服':
                girl.append(line_spoken)
        else:
            save_file(boy, girl, count)
            boy = []
            girl = []
            count += 1
    save_file(boy, girl, count)
    f.close()
split_file('record.txt')
```

## 8.2 文件系统：介绍一个高大上的东西



接下来会介绍跟 Python 的文件相关的一些十分有用的模块。模块是什么？其实我们写的每一个源代码文件（*.py）都是一个模块。Python 自身带有非常多实用的模块，在日常编程中，如果能够熟练地掌握它们，将事半功倍。

比如刚开始介绍的文字小游戏，里边就用 random 模块的 randint()函数来生成随机数。然而要使用这个 randint()函数，直接就调用可不行：

```
>>> random.randint(0, 9)
Traceback (most recent call last):
  File "<pyshell#140>", line 1, in <module>
    random.randint(0, 9)
NameError: name 'random' is not defined
```

正确的做法应该是先使用 import 语句导入模块，然后再使用：

```
>>> import random
>>> random.randint(0, 9)
3
>>> random.randint(0, 9)
1
>>> random.randint(0, 9)
8
```

首先要介绍的是高大上的 OS 模块，OS 就是 Operating System 的缩写，意思是操作系统，而平时经常说 iOS 就是 iPhone OS 的意思，即苹果手机的操作系统。但这里小甲鱼说 OS 模块高大上，并不是因为跟苹果或土豪金拉边才这么说。之所以说 OS 模块高大上，是因为对于

文件系统的访问，Python一般是通过OS模块来实现的。我们所知道常用的操作系统就有Windows、Mac OS、Linux、UNIX等，这些操作系统底层对于文件系统的访问工作原理是不一样的，因此你可能就要针对不同的系统来考虑使用哪些文件系统模块……这样的做法是非常不友好且麻烦的，因为这意味着当你的程序运行环境一旦改变，你就要相应地去修改大量的代码来应付。

但是Python是跨平台的语言，也就是说，同样的源代码在不同的操作系统不需要修改就可以同样实现。有了OS模块，不需要关心什么操作系统下使用什么模块，OS模块会帮你选择正确的模块并调用。

表8-3列举了OS模块中关于文件/目录常用的函数使用方法。

**表8-3　OS模块中关于文件/目录常用的函数使用方法**

| 函　数　名 | 使用方法 |
|---|---|
| getcwd() | 返回当前工作目录 |
| chdir(path) | 改变工作目录 |
| listdir(path='.') | 列举指定目录中的文件名('.'表示当前目录，'..'表示上一级目录) |
| mkdir(path) | 创建单层目录，如该目录已存在抛出异常 |
| makedirs(path) | 递归创建多层目录，如果该目录已存在则抛出异常，注意：'E:\\a\\b'和'E:\\a\\c'并不会冲突 |
| remove(path) | 删除文件 |
| rmdir(path) | 删除单层目录，如果该目录非空则抛出异常 |
| removedirs(path) | 递归删除目录，从子目录到父目录逐层尝试删除，遇到目录非空则抛出异常 |
| rename(old, new) | 将文件old重命名为new |
| system(command) | 运行系统的shell命令 |
| 以下是支持路径操作中常用到的一些定义，支持所有平台 | |
| os.curdir | 指代当前目录('.') |
| os.pardir | 指代上一级目录('..') |
| os.sep | 输出操作系统特定的路径分隔符(在Windows下为'\\'，Linux下为'/') |
| os.linesep | 当前平台使用的行终止符(在Windows下为'\r\n'，Linux下为'\n') |
| os.name | 指代当前使用的操作系统(包括'posix'、'nt'、'mac'、'os2'、'ce'、'java') |

### 1. getcwd()

在有些情况下我们需要获得应用程序当前的工作目录(比如要保存临时文件)，那么可以使用getcwd()函数获得：

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
```

### 2. chdir(path)

用chdir()函数可以改变当前工作目录，比如可以切换到E盘：

```
>>> os.chdir("E:\\")
>>> os.getcwd()
'E:\\'
```

### 3. listdir(path='.')

有时候你可能需要知道当前目录下有哪些文件和子目录，那么listdir()函数可以帮你列举出来。path参数用于指定列举的目录，默认值是'.'，代表根目录，也可以使用'..'代表上一层目录：

```
>>> os.listdir()
['$RECYCLE.BIN', 'Arduino', 'System Volume Information', '工作室', '工具箱', '鱼C光盘', '鱼C工作室编程教学']
>>> os.listdir("C:\\")
['$360Section', '$Recycle.Bin', '360SANDBOX', 'Boot', 'bootmgr', 'BOOTNXT', 'DkHyperbootSync', 'Documents and Settings', 'hiberfil.sys', 'Intel', 'iSee', 'mfg', 'MSOCache', 'OneDriveTemp', 'pagefile.sys', 'PerfLogs', 'Program Files', 'Program Files (x86)', 'ProgramData', 'Python27', 'Python34', 'Recovery', 'Recovery.txt', 'swapfile.sys', 'System Volume Information', 'Users', 'Windows']
```

### 4. mkdir(path)

mkdir()函数用于创建文件夹，如果该文件夹存在，则抛出FileExistsError异常：

```
>>> os.mkdir("test")
>>> os.listdir()
['$RECYCLE.BIN', 'Arduino', 'System Volume Information', 'test', '工作室', '工具箱', '鱼C光盘', '鱼C工作室编程教学']
>>> os.mkdir("test")
Traceback (most recent call last):
  File "<pyshell#156>", line 1, in <module>
    os.mkdir("test")
FileExistsError: [WinError 183] 当文件已存在时，无法创建该文件.: 'test'
```

### 5. makedirs(path)

makedirs()函数可以用于创建多层目录：

```
>>> os.makedirs(r".\a\b\c")
```

效果如图8-2所示。




图8-2 makedirs()函数

### 6. remove(path)、rmdir(path)和removedirs(path)

remove()函数用于删除指定的文件，注意是删除文件，不是删除目录。如果要删除目录，则用rmdir()函数；如果要删除多层目录，则用removedirs()函数。

```
>>> os.listdir()
['a', 'b', 'test.txt']
>>> # 当前工作目录结构为 a\b\c,b\,test.txt
>>> os.remove("test.txt")
>>> os.rmdir("b")
>>> os.removedirs(r"a\b\c")
>>> os.listdir()
[]
```

### 7. rename(old, new)

rename()函数用于重命名文件或文件夹：

```
>>> os.listdir()
['a', 'a.txt']
>>> os.rename("a", "b")
>>> os.rename("a.txt", "b.txt")
>>> os.listdir()
['b', 'b.txt']
```

### 8. system(command)

几乎每个操作系统都会提供一些小工具，system()函数用于使用这些小工具：

```
>>> os.system("calc") # calc 是 Windows 系统自带的计算器
```

回车后即弹出计算器，效果如图 8-3 所示。




图 8-3 system()函数

### 9. walk(top)

最后是 walk()函数，这个函数在有些时候确实非常有用，可以省去你很多麻烦。该函数的作用是遍历 top 参数指定路径下的所有子目录，并将结果返回一个三元组(路径，[包含目录]，[包含文件])。来看下面的例子：

```
>>> for i in os.walk("test"):
	print(i)
('test', ['a', 'b', 'c'], [])
('test\\a', [], ['a.txt'])
('test\\b', ['b1', 'b2'], ['b.txt'])
('test\\b\\b1', [], ['b1.txt'])
```

```
('test\\b\\b2', [], ['b2.txt'])
('test\\c', ['c1'], [])
('test\\c\\c1', ['c11'], [])
('test\\c\\c1\\c11', [], ['c11.txt'])
```

为了便于理解，我画个实际的文件夹分布图给大家对比一下，如图 8-4 所示。

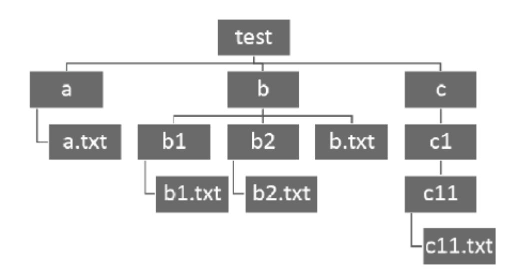



图 8-4　walk()函数

另外 path 模块还提供了一些很实用的定义，分别是：os.curdir 表示当前目录；os.pardir 表示上一级目录('..')；os.sep 表示路径的分隔符，比如 Windows 系统下为'\\'，Linux 下为'/'；os.linesep 表示当前平台使用的行终止符(在 Windows 下为'\r\n'，Linux 下为'\n')；os.name 表示当前使用的操作系统。

另一个强大的模块是 os.path，它可以完成一些针对路径名的操作。表 8-4 列举了 os.path 中常用到的函数使用方法。

**表 8-4　os.path 模块中关于路径常用的函数使用方法**

| 函　数　名 | 使 用 方 法 |
|---|---|
| basename(path) | 去掉目录路径，单独返回文件名 |
| dirname(path) | 去掉文件名，单独返回目录路径 |
| join(path1[,path2[, ...]]) | 将 path1 和 path2 各部分组合成一个路径名 |
| split(path) | 分割文件名与路径，返回(f_path, f_name)元组。如果完全使用目录，它也会将最后一个目录作为文件名分离，且不会判断文件或者目录是否存在 |
| splitext(path) | 分离文件名与扩展名，返回(f_name,f_extension)元组 |
| getsize(file) | 返回指定文件的尺寸，单位是字节 |
| getatime(file) | 返回指定文件最近的访问时间(浮点型秒数，可用 time 模块的 gmtime()或 localtime()函数换算) |
| getctime(file) | 返回指定文件的创建时间(浮点型秒数，可用 time 模块的 gmtime()或 localtime()函数换算) |
| getmtime(file) | 返回指定文件最新的修改时间(浮点型秒数，可用 time 模块的 gmtime()或 localtime()函数换算) |
| 以下为函数返回 True 或 False | |
| exists(path) | 判断指定路径(目录或文件)是否存在 |
| isabs(path) | 判断指定路径是否为绝对路径 |
| isdir(path) | 判断指定路径是否存在且是一个目录 |
| isfile(path) | 判断指定路径是否存在且是一个文件 |
| islink(path) | 判断指定路径是否存在且是一个符号链接 |
| ismount(path) | 判断指定路径是否存在且是一个挂载点 |
| samefile(path1, path2) | 判断 path1 和 path2 两个路径是否指向同一个文件 |

### 10. basename(path)和 dirname(path)

basename()和 dirname()函数分别用于获得文件名和路径名：

```
>>> os.path.dirname(r"a\b\test.txt")
'a\\b'
>>> os.path.basename(r"a\b\text.txt")
'text.txt'
```

### 11. join(path1[,path2[,...]])

join()函数跟 BIF 的那个 join()函数不同，os.path.join()是用于将路径名和文件名组合成一个完整的路径：

```
>>> os.path.join(r"C:\Python34\Test", "FishC.txt")
'C:\\Python34\\Test\\FishC.txt'
```

### 12. split(path)和 splitext(path)

split()和 splitext()函数都用于分割路径，split()函数分割路径和文件名(如果完全使用目录，它也会将最后一个目录作为文件名分离，且不会判断文件或者目录是否存在)；splitext()函数则是用于分割文件名和扩展名：

```
>>> os.path.split(r"a\b\test.txt")
('a\\b', 'test.txt')
>>> os.path.splitext(r"a\b\test.txt")
('a\\b\\test', '.txt')
```

### 13. getsize(file)

getsize()函数用于获取文件的尺寸，返回值是以字节为单位：

```
>>> os.chdir(r"C:\Python34")
>>> os.path.getsize("python.exe")
40960
```

### 14. getatime(file)、getctime(file)和 getmtime(file)

getatime()、getctime()和 getmtime()分别用于获得文件的最近访问时间、创建时间和修改时间。不过返回值是浮点型秒数，可用 time 模块的 gmtime()或 localtime()函数换算：

```
>>> import time
>>> temp = time.localtime(os.path.getatime("python.exe"))
>>> print("python.exe 被访问的时间是: ", time.strftime("%d %b %Y %H:%M:%S", temp))
python.exe 被访问的时间是: 27 May 2015 21:16:59
>>> temp = time.localtime(os.path.getctime("python.exe"))
>>> print("python.exe 被创建的时间是: ", time.strftime("%d %b %Y %H:%M:%S", temp))
python.exe 被创建的时间是: 24 Feb 2015 22:44:44
>>> temp = time.localtime(os.path.getmtime("python.exe"))
>>> print("python.exe 被修改的时间是: ", time.strftime("%d %b %Y %H:%M:%S", temp))
```



```
python.exe 被修改的时间是: 24 Feb 2015 22:44:44
```

还有一些函数返回布尔类型的值，具体的解释见表 8-4，这里就不一一举例了。

## 8.3 pickle：腌制一缸美味的泡菜



从一个文件里读取字符串非常简单，但如果想要读取出数值，那就需要多费点儿周折。因为无论是 read()方法，还是 readline()方法，都是返回一个字符串，如果希望从字符串里边提取出数值的话，可以使用 int()函数或 float()函数把类似'123'或'3.14'这类字符串强制转换为具体的数值。

此前一直在讲保存文本，然而当要保存的数据像列表、字典甚至是类的实例这些更复杂的数据类型时，普通的文件操作就会变得不知所措。也许你会把这些都转换为字符串，再写入到一个文本文件中保存起来，但是很快你就会发现要把这个过程反过来，从文本文件恢复数据对象，就变得异常麻烦了。

所幸的是，Python 提供了一个标准模块，使用这个模块，就可以非常容易地将列表、字典这类复杂数据类型存储为文件了。这个模块就是本节要介绍的 pickle 模块。

pickle 就是泡菜，腌菜的意思，相信很多女读者都对韩国泡菜尤其情有独钟。至于 Python 的作者为何把这么一个高大上模块命名为泡菜，我想应该是跟韩剧脱不了干系。

好，说回这个泡菜。用官方文档中的话说，这是一个令人惊叹(amazing)的模块，它几乎可以把所有 Python 的对象都转化为二进制的形式存放，这个过程称为 pickling，那么从二进制形式转换回对象的过程称为 unpickling。

说了这么多，还是来点干货吧：

```
# p8_3.py
import pickle

my_list = [123, 3.14, '小甲鱼', ['another list']]
pickle_file = open('E:\\my_list.pkl', 'wb')
pickle.dump(my_list, pickle_file)
pickle_file.close()
```

分析一下：这里希望把这个列表永久保存起来(保存成文件)，打开的文件一定要以二进制的形式打开，后缀名倒是可以随意，不过既然是使用 pickle 保存，为了今后容易记忆，建议还是使用.pkl 或.pickle。使用 dump 方法来保存数据，完成后记得保存，跟操作普通文本文件一样。

程序执行之后 E 盘会出现一个 my_list.pkl 的文件，用记事本打开之后显示乱码(因为它保存的是二进制形式)，如图 8-5 所示。

那么在使用的时候只需用二进制模式先把文件打开，然后用 load 把数据加载进来：

```
# p8_4.py
import pickle

pickle_file = open("E:\\my_list.pkl", "rb")
my_list = pickle.load(pickle_file)
print(my_list)
```

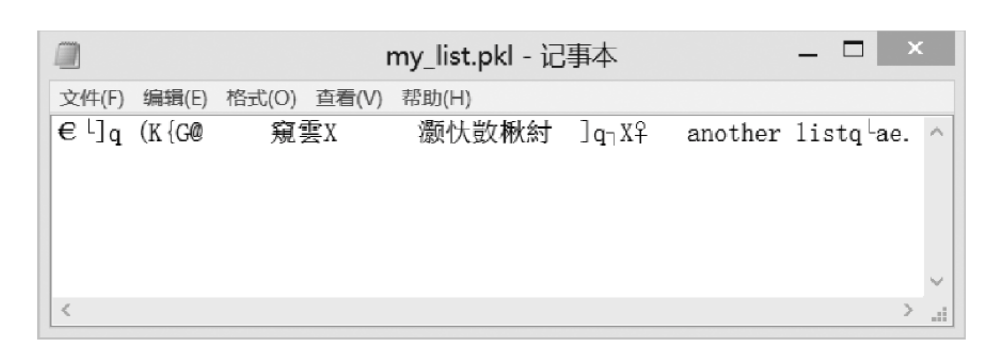



图 8-5　保存为 pickle 文件

程序执行后又取回我们的列表啦：

```
>>>
[123, 3.14, '小甲鱼', ['another list']]
>>>
```

利用 pickle 模块，不仅可以保存列表，事实上 pickle 可以保存任何你能想象得到的东西。



# 第9章 异常处理



## 9.1 你不可能总是对的

因为我们是人，不是神，所以我们经常会犯错。当然程序员也不例外，就算是经验丰富的码农，也不能保证写出来的代码百分之百没有任何问题(要不哪来那么多 0Day 漏洞)。另外，作为一个合格的程序员，在编程的时候一定要意识到一点，就是永远不要相信你的用户。要把他们想象成熊孩子，把他们想象成黑客，这样你写出来的程序自然会更加安全和稳定。

那么既然程序总会出问题，我们就应该学会用适当的方法去解决问题。程序出现逻辑错误或者用户输入不合法都会引发异常，但这些异常并不是致命的，不会导致程序崩溃死掉。可以利用 Python 提供的异常处理机制，在异常出现的时候及时捕获，并从内部自我消化掉。

那么什么是异常呢？举个例子：

```
# p9_1.py
file_name = input('请输入要打开的文件名: ')
f = open(file_name, 'r')
print('文件的内容是: ')

for each_line in f:
    print(each_line)
```

这里当然假设用户的输入是正确的，但只要用户输入一个不存在的文件名，那么上面的代码就不堪一击：

```
>>>
请输入要打开的文件名: 我为什么是一个文档.txt
Traceback (most recent call last):
  File "E:\p9_1.py", line 2, in <module>
    f = open(file_name, 'r')
FileNotFoundError: [Errno 2] No such file or directory: '我为什么是一个文档.txt'
```

上面的例子就抛出了一个 FileNotFoundError 异常，那 Python 通常还可能抛出哪些异常呢？这里给大家做个总结，今后遇到这样的异常时就不会感觉到陌生了。

### 1. AssertionError：断言语句(assert)失败

大家还记得断言语句吧？在关于分支和循环的章节里讲过。当 assert 这个关键字后边的条件为假的时候，程序将停止并抛出 AssertionError 异常。assert 语句一般是在测试程序的

时候用于在代码中置入检查点：

```
>>> my_list = ["小甲鱼"]
>>> assert len(my_list) > 0
>>> my_list.pop()
'小甲鱼'
>>> assert len(my_list) > 0
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    assert len(my_list) > 0
AssertionError
```

#### 2. AttributeError：尝试访问未知的对象属性

当试图访问的对象属性不存在时抛出的异常：

```
>>> my_list = []
>>> my_list.fishc
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    my_list.fishc
AttributeError: 'list' object has no attribute 'fishc'
```

#### 3. IndexError：索引超出序列的范围

在使用序列的时候就常常会遇到 IndexError 异常，原因是索引超出序列范围的内容：

```
>>> my_list = [1, 2, 3]
>>> my_list[3]
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    my_list[3]
IndexError: list index out of range
```

#### 4. KeyError：字典中查找一个不存在的关键字

当试图在字典中查找一个不存在的关键字时就会引发 KeyError 异常，因此建议使用 dict.get()方法：

```
>>> my_dict = {"one":1, "two":2, "three":3}
>>> my_dict["one"]
1
>>> my_dict["four"]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    my_dict["four"]
KeyError: 'four'
```

#### 5. NameError：尝试访问一个不存在的变量

当尝试访问一个不存在的变量时，Python 会抛出 NameError 异常：

```
>>> fishc
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    fishc
NameError: name 'fishc' is not defined
```

### 6. OSError：操作系统产生的异常

OSError 顾名思义就是操作系统产生的异常，像打开一个不存在的文件会引发 FileNotFoundError，而这个 FileNotFoundError 就是 OSError 的子类。例子上面已经演示过，这里就不再重复。

### 7. SyntaxError：Python 的语法错误

如果遇到 SyntaxError 是 Python 的语法错误，这时 Python 的代码并不能继续执行，你应该先找到并改正错误：

```
>>> print "I love fishc.com"
SyntaxError: Missing parentheses in call to 'print'
```

### 8. TypeError：不同类型间的无效操作

有些类型不同是不能相互进行计算的，否则会抛出 TypeError 异常：

```
>>> 1 + "1"
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    1 + "1"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

### 9. ZeroDivisionError：除数为零

地球人都知道除数不能为零，所以除以零就会引发 ZeroDivisionError 异常：

```
>>> 5 / 0
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    5 / 0
ZeroDivisionError: division by zero
```

好了，知道程序抛出异常就说明这个程序有问题，但问题并不致命，所以可以通过捕获这些异常，并纠正这些错误就行。那应该如何捕获和处理异常呢？

异常捕获可以使用 try 语句来实现，任何出现在 try 语句范围内的异常都会被及时捕获到。try 语句有两种实现形式：一种是 try-except，另一种是 try-finally。

## 9.2 try-except 语句



try-except 语句格式如下：

```
try:
```

```
    检测范围
except Exception[as reason]:
    出现异常(Exception)后的处理代码
```

try-except 语句用于检测和处理异常，举个例子来说明这一切是如何工作的：

```
# p9_2.py
f = open('我为什么是一个文档.txt')
print(f.read())
f.close()
```

以上代码在“我为什么是一个文档.txt”这个文档不存在的时候，Python 就会报错说文件不存在：

```
>>>
Traceback (most recent call last):
  File "E:\p9_2.py", line 1, in <module>
    f = open('我为什么是一个文档.txt')
FileNotFoundError: [Errno 2] No such file or directory: '我为什么是一个文档.txt'
>>>
```

显然这样的用户体验不好，因此可以这么修改：

```
# p9_3.py
try:
    f = open('我为什么是一个文档.txt')
    print(f.read())
    f.close()
except OSError:
    print('文件打开的过程中出错啦 T_T')
```

上面的例子由于使用了大家习惯的语言来表述错误信息，用户体验当然会好很多：

```
>>>
文件打开的过程中出错啦 T_T
>>>
```

但是从程序员的角度来看，导致 OSError 异常的原因有很多(例如 FileExistsError、FileNotFoundError、PermissionError 等等)，所以可能会更在意错误的具体内容，这里可以使用 as 把具体的错误信息给打印出来：

```
except OSError as reason:
    print('文件出错啦 T_T\n 错误原因是: ' + str(reason))
```

## 9.2.1 针对不同异常设置多个 except

一个 try 语句还可以和多个 except 语句搭配，分别对感兴趣的异常进行检测处理：

```
# p9_4.py
try:
    sum = 1 + '1'
    f = open('我是一个不存在的文档.txt')
    print(f.read())
    f.close()
```

```
except OSError as reason:
    print('文件出错啦 T_T\n 错误原因是: ' + str(reason))
except TypeError as reason:
    print('类型出错啦 T_T\n 错误原因是: ' + str(reason))
```

### 9.2.2 对多个异常统一处理

except 后边还可以跟多个异常,然后对这些异常进行统一的处理:

```
# p9_5.py
try:
    int('abc')
    sum = 1 + '1'
    f = open('我是一个不存在的文档.txt')
    print(f.read())
    f.close()
except (OSError, TypeError):
    print('出错啦 T_T\n 错误原因是: ' + str(reason))
```

### 9.2.3 捕获所有异常

如果你无法确定要对哪一类异常进行处理,只是希望在 try 语句块里一旦出现任何异常,可以给用户一个"看得懂"的提醒,那么可以这么做:

```
…
except:
    print('出错啦～')
…
```

不过通常不建议你这么做,因为它会隐藏所有程序员未想到并且未做好处理准备的错误,例如当用户输入 Ctrl+C 试图终止程序,却被解释为 KeyboardInterrupt 异常。另外要注意的是,try 语句检测范围内一旦出现异常,剩下的语句将不会被执行。

## 9.3 try-finally 语句

如果"我是一个不存在的文档"确实存在,open()函数正常返回文件对象,但异常却发生在成功打开文件后的 sum = 1 + '1'语句上。此时 Python 将直接跳到 except 语句,也就是说,文件打开了,但并没有执行关闭文件的命令:

```
# p9_6.py
try:
    f = open('我是一个不存在的文档.txt')
print(f.read())
    sum = 1 + '1'
    f.close()
except:
    print('出错啦')
```

为了实现像这种"就算出现异常,但也不得不执行的收尾工作(比如在程序崩溃前保存用

户文档)”,引入了 finally 来扩展 try:

```
# p9_7.py
try:
    f = open('我是一个不存在的文档.txt')
print(f.read())
    sum = 1 + '1'
except:
    print('出错啦')
finally:
    f.close()
```

如果 try 语句块中没有出现任何运行时错误,会跳过 except 语句块执行 finally 语句块的内容。如果出现异常,则会先执行 except 语句块的内容再执行 finally 语句块的内容。总之,finally 语句块中的内容就是确保无论如何都将被执行的内容。

## 9.4 raise 语句

有读者可能会问,我的代码能不能自己抛出一个异常呢?答案是可以的,你可以使用 raise 语句抛出一个异常:

```
>>> raise ZeroDivisionError
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise ZeroDivisionError
ZeroDivisionError
```

抛出的异常还可以带参数,表示异常的解释:

```
>>> raise ZeroDivisionError("除数不能为零!")
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    raise ZeroDivisionError("除数不能为零!")
ZeroDivisionError: 除数不能为零!
```

## 9.5 丰富的 else 语句



有读者可能会说,else 语句还有啥好讲的?经常跟 if 语句进行搭配用于条件判断嘛。没错,对于大多数编程语言来说,else 语句都只能跟 if 语句搭配。但在 Python 里,else 语句的功能更加丰富。

在 Python 中,else 语句不仅能跟 if 语句搭,构成“要么怎样,要么不怎样”的句式;它还能跟循环语句(for 语句或者 while 语句),构成“干完了能怎样,干不完就别想怎样”的句式;其实 else 语句还能够跟刚刚讲的异常处理进行搭配,构成“没有问题?那就干吧”的句式,下边逐个给大家解释。

### 1. 要么怎样,要么不怎样

典型的 if-else 搭配:

```
if 条件:
    条件为真执行
else:
    条件为假执行
```

**2. 干完了能怎样，干不完就别想怎样**

else 可以跟 for 和 while 循环语句配合使用，但 else 语句块只在循环完成后执行，也就是说，如果循环中间使用 break 语句跳出循环，那么 else 里边的内容就不会被执行了。举个例子：

```
# p9_8.py
def showMaxFactor(num):
    count = num // 2
    while count > 1:
        if num % count == 0:
            print('%d 最大的约数是 %d' % (num, count))
            break
        count -= 1
    else:
        print('%d 是素数!' % num)

num = int(input('请输入一个数: '))
showMaxFactor(num)
```

这个小程序主要是求用户输入的数的最大约数，如果是素数的话就顺便提醒“这是一个素数”。注意要使用地板除法(count = num // 2)哦，否则结果会出错。使用暴力的方法一个个尝试(num % count == 0)，如果符合条件则打印出最大的约数，并 break，同时不会执行 else 语句块的内容了。但如果一直没有遇到合适的条件，则会执行 else 语句块内容。

for 语句的用法跟 while 一样，这里就不重复举例了。

**3. 没有问题？那就干吧**

else 语句还能跟刚刚学的异常处理进行搭配，实现跟与循环语句搭配差不多：只要 try 语句块里没有出现任何异常，那么就会执行 else 语句块里的内容啦。举个例子：

```
# p9_9.py
try:
    int('abc')
except ValueError as reason:
    print('出错啦: ' + str(reason))
else:
    print('没有任何异常!')
```

## 9.6　简洁的 with 语句

有读者可能觉着打开文件又要关闭文件，还要关注异常处理有点烦人，所以 Python 提供了一个 with 语句，利用这个语句抽象出文件操作中频繁使用的 try/except/finally 相关的细

节。对文件操作使用 with 语句,将大大减少代码量,而且你再也不用担心出现文件打开了忘记关闭的问题了(with 会自动帮你关闭文件)。举个例子:

```
# p9_10.py
try:
    f = open('data.txt', 'w')
    for each_line in f:
        print(each_line)
except OSError as reason:
    print('出错啦: ' + str(reason))
finally:
    f.close()
```

使用 with 语句,可以改成这样:

```
# p9_11.py
try:
    with open('data.txt', 'w') as f:
        for each_line in f:
            print(each_line)
except OSError as reason:
print('出错啦: ' + str(reason))
```

是不是很方便呢?有了 with 语句,就再也不用担心忘记关闭文件了。

# 第10章 图形用户界面入门

本章给大家介绍图形用户界面编程，也就是平时常说的 GUI(Graphical User Interface，读作[gu:i])编程，那些带有按钮、文本、输入框的窗口的编程，相信大家都不会陌生。

目前有很多 Python 的 GUI 工具包可供选择，Python 有一个非常简单的 GUI 工具包：EasyGui。EasyGui 跟它的名字一样简单，一旦你的模块导入 EasyGui，GUI 操作就是一个简单地调用 EasyGui 函数的几个参数的问题了。

EasyGui 官网：http://easygui.sourceforge.net。

本书配套资源：easygui-0.96.zip。

使用标准方法安装：

- 解压 easygui-0.96.zip。
- 使用命令窗口切换到 easygui-docs-0.96 的目录下。
- 在 Windows 下执行 C:\Python34\python.exe setup.py install。
- 在 Linux 或 Mac 下执行 sudo /usr/bin/python34 setup.py install。

Windows 下的安装界面如图 10-1 所示。

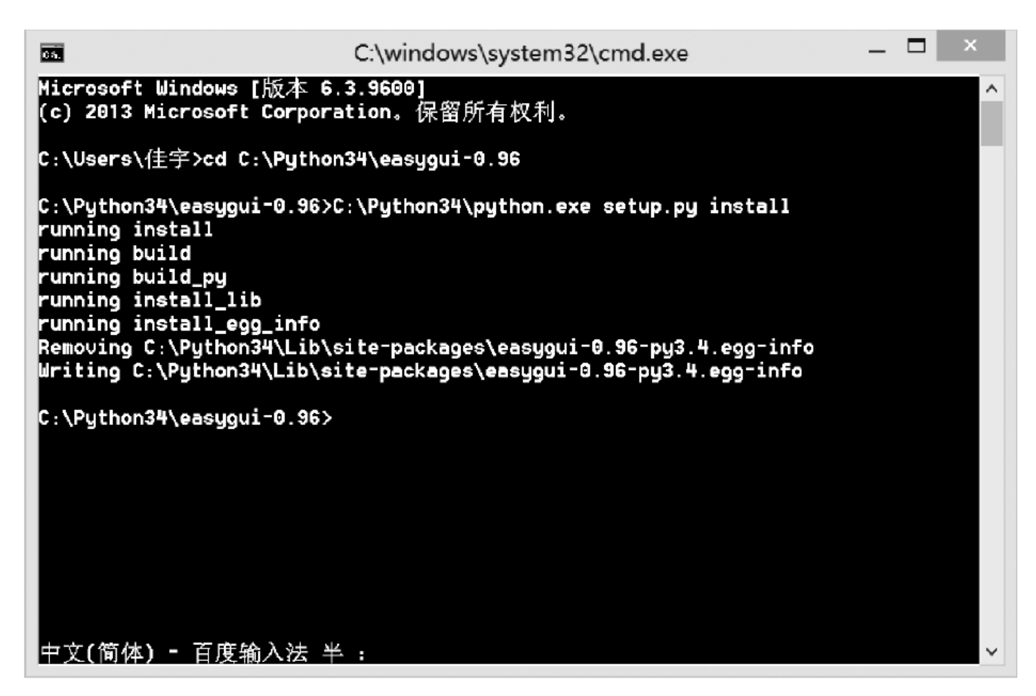



图 10-1 EasyGui 的安装

希望深入学习 Python 的读者，可以在本书配套资源中的 easygui-0.96.zip 压缩包中找到 EasyGui 各个函数的实现源代码（下载地址 http://bbs.fishc.com/thread-46069-1-1.html）。

附件：easygui-0.96.zip。

## 10.1 导入 EasyGui

为了使用 EasyGui 这个模块，你应该先导入它。最简单的导入语句是 import easygui。

如果使用这种形式导入的话，那么在使用 EasyGui 的函数的时候，必须在函数的前面加上前缀 easygui：

```
>>> import easygui
>>> easygui.msgbox("嗨，大家好～")
```

回车后即弹出消息框，如图 10-2 所示。




图 10-2　导入 EasyGui 模块（方法一）

另一种选择是导入整个 EasyGui 包：from easygui import *，这样使得我们更容易调用 EasyGui 的函数，可以直接这样编写代码：

```
>>> from easygui import *
>>> msgbox("嗨，小美女～")
```

回车后即弹出消息框，如图 10-3 所示。

第三种方案是使用类似下边的 import 语句（建议使用）：import easygui as g，这样可以让你保持 EasyGui 的命名空间，同时减少输入字符的数量：

```
>>> import easygui as g
>>> g.msgbox("嗨，鱼 C～")
```

回车后即弹出消息框，如图 10-4 所示。




图 10-3　导入 EasyGui 模块（方法二）

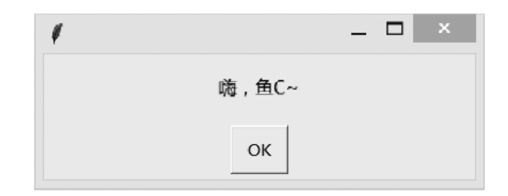



图 10-4　导入 EasyGui 模块（方法三）

## 10.2 使用 EasyGui

举一个简单的例子：

```
# p10_1.py
import easygui as g
import sys

while 1:
```

```
g.msgbox("嗨,欢迎进入第一个界面小游戏^_^")
msg ="请问你希望在鱼C工作室学习到什么知识呢?"
title = "小游戏互动"
choices = ["谈恋爱", "编程", "demo", "琴棋书画"]
choice = g.choicebox(msg, title, choices)
# note that we convert choice to string, in case
# the user cancelled the choice, and we got None.
g.msgbox("你的选择是: " + str(choice), "结果")
msg = "你希望重新开始小游戏吗?"
title = "请选择"
if g.ccbox(msg, title): # show a Continue/Cancel dialog
        pass # user chose Continue
else:
        sys.exit(0) # user chose Cancel
```

实现过程如图10-5～图10-8所示。

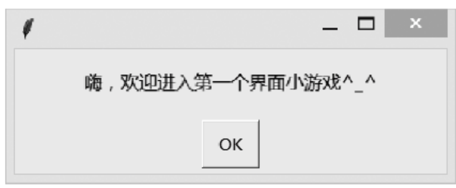



图10-5　使用EasyGui编写第一个界面小游戏(一)




图10-6　使用EasyGui编写第一个界面小游戏(二)




图10-7　使用EasyGui编写第一个界面小游戏(三)

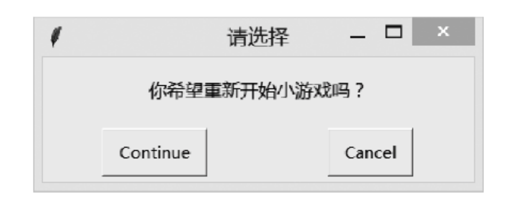



图10-8　使用EasyGui编写第一个界面小游戏(四)

## 10.3 修改默认设置

默认情况下显示的对话框非常大，而且字体也相对难看。这里可以手动调整 EsayGui 的参数修改。

修改位置为 C:\Python34\Lib\site-packages\easygui.py。

更改对话框尺寸：找到 def __choicebox，下边的 root_width = int((screen_width * 0.8))和 root_height = int((screen_height * 0.5))分别改为 root_width = int((screen_width * 0.4))和 root_height = int((screen_height * 0.25))。

更改字体：找到 PROPORTIONAL_FONT_FAMILY = ("MS", "Sans", "Serif")改为 PROPORTIONAL_FONT_FAMILY = ("微软雅黑")。

EasyGui 提供了非常多的组件供我们实现一个完整的界面程序，刚才给大家演示的就是 msgbox、choicebox 和 ccbox 的用法。关于更多的组件使用，大家可以参考小甲鱼翻译改编的《EasyGui 学习文档》：http://bbs.fishc.com/thread-46069-1-1.html。

# 第11章 类和对象

## 11.1 给大家介绍对象



很多读者此前肯定听说过 Python 无处不对象，然而他们并不知道对象到底是个什么东西。他们只是在学习的时候听说过有面向对象编程……这就像学开车，你并不用理解汽车为什么会跑，但作为赛车手，这些原理就必须要懂，因为这有助于他把车开得更好。因此，本章就向大家隆重地介绍对象！

大家之前已经听说过封装的概念，把乱七八糟的数据扔进列表里边，这是一种封装，是数据层面的封装；把常用的代码段打包成一个函数，这也是一种封装，是语句层面的封装；本章学习的对象，也是一种封装的思想，不过这种思想显然要更先进一步：对象的来源是模拟真实世界，把数据和代码都封装在了一起。

打个比方，乌龟就是真实世界的一个对象，那么通常应该如何来描述这个对象呢？是不是把它分为两部分来说？

(1) 可以从静态的特征来描述，例如，绿色的、有四条腿，10kg 重，有外壳，还有个大嘴巴，这是静态一方面的描述。

(2) 还可以从动态的行为来描述，例如说它会爬，你如果追它，它就会跑，然后你把它逼急了，它就会咬人，被它咬到了，据说要打雷才会松开嘴巴……它的嘴巴的重要作用不是用来咬人，是用来吃东西的，然后它还会睡觉。这些都是从行为方面进行描述的。

那如果把一个人作为对象，你会从哪两方面来描述这个人？

对嘛，无非就是他长什么样？这是从外观方面找特征，例如，眼歪鼻子斜、胸大屁股翘，这些都是静态的特征。另一方面就是描述他的行为？例如，唱歌、街舞、打篮球等等。

## 11.2 对象＝属性＋方法

Python 中的对象也是如此，一个对象的特征称为“属性”，一个对象的行为称为“方法”。

如果把“乌龟”写成代码，将会是下边这样：

```
# p11_1.py
class Turtle:
    # Python 中的类名约定以大写字母开头
    # 特征的描述称为属性，在代码层面来看其实就是变量
    color = 'green'
```

```
    weight = 10
    legs = 4
    shell = True
    mouth = '大嘴'

    # 方法实际就是函数,通过调用这些函数来完成某些工作
    def climb(self):
        print("我正在很努力地向前爬…")
    def run(self):
        print("我正在飞快地向前跑…")
    def bite(self):
        print("咬死你咬死你!!")
    def eat(self):
        print("有得吃,真满足^_^")
    def sleep(self):
        print("困了,睡了,晚安,Zzzz")
```

以上代码定义了对象的特征(属性)和行为(方法),但还不是一个完整的对象,将定义的这些称为类(Class)。需要使用类来创建一个真正的对象,这个对象就叫作这个类的一个实例(Instance),也叫实例对象(Instance Objects)。

有些读者可能还不大理解,你可以这么想:这就好比工厂的流水线要生产一系列玩具,是不是要先做出这个玩具的模具,然后根据这个模具再进行批量生产,才得到真正的玩具?

再举个例子:盖房子,是不是先要有个图纸,但光有个图纸你能不能住进去?显然不能,图纸只能告诉你这个房子长什么样,但图纸并不是真正的房子。要根据图纸用钢筋水泥建造出来的房子才能住人,另外根据一张图纸就能盖出很多的房子。

好,说了这么多,那真正的实例对象怎么创建?

创建一个对象,也叫类的实例化,其实非常简单:

```
>>> # 先运行 p11_1.py
>>> tt = Turtle()
>>>
```

注意,类名后边跟着的小括号,这跟调用函数是一样的,所以在Python中,类名约定用大写字母开头,函数用小写字母开头,这样更容易区分。另外赋值操作并不是必需的,但如果没有把创建好的实例对象赋值给一个变量,那这个对象就没办法使用,因为没有任何引用指向这个实例,最终会被Python的垃圾收集机制自动回收。

那如果要调用对象里的方法,使用点操作符(.)即可,其实我们已经用了何止千百遍:

```
>>> tt.climb()
我正在很努力地向前爬…
>>> tt.bite()
咬死你咬死你!!
>>> tt.sleep()
困了,睡了,晚安,Zzzz
```

## 11.3 面向对象编程



经过前边的热身,相信大家对类和对象已经有了初步的认识,但似乎还是懵懵懂懂:好像面向对象编程很厉害,但不知道具体怎么用?下面通过几个主题,尝试给大家进一步剖析

Python 的类和对象。

### 11.3.1　self 是什么

细心的读者会发现对象的方法都会有一个 self 参数，那这个 self 到底是个什么东西呢？如果此前接触过其他面向对象的编程语言，例如 C++，那么你应该很容易对号入座，Python 的 self 其实就相当于 C++的 this 指针。

这里为了照顾大部分的初学编程的读者，讲解下 self 到底是个什么东西。如果把类比作是图纸，那么由类实例化后的对象才是真正可以住的房子。根据一张图纸就可以设计出成千上万的房子，它们长得都差不多，但它们都有不同的主人。每个人都只能回自己的家里，陪伴自己的孩子……所以 self 这里就相当于每个房子的门牌号，有了 self，你就可以轻松找到自己的房子。

Python 的 self 参数就是同一个道理，由同一个类可以生成无数对象，当一个对象的方法被调用的时候，对象会将自身的引用作为第一个参数传给该方法，那么 Python 就知道需要操作哪个对象的方法了。

通过一个例子稍微感受下：

```
>>> class Ball:
	def setName(self, name):
		self.name = name
	def kick(self):
		print("我叫 %s,噢～谁踢我?!" % self.name)

>>> a = Ball()
>>> a.setName("飞火流星")
>>> b = Ball()
>>> b.setName("团队之星")
>>> c = Ball()
>>> c.setName("土豆") # 乱入…
>>> a.kick()
我叫飞火流星,噢～谁踢我?!
>>> b.kick()
我叫团队之星,噢～谁踢我?!
>>> c.kick()
我叫土豆,噢～谁踢我?!
```

### 11.3.2　你听说过 Python 的魔法方法吗

据说，Python 的对象天生拥有一些神奇的方法，它们是面向对象的 Python 的一切。它们是可以给你的类增加魔力的特殊方法，如果你的对象实现了这些方法中的某一个，那么这个方法就会在特殊的情况下被 Python 所调用，而这一切都是自动发生的。

Python 的这些具有魔力的方法，总是被双下划线所包围，今天就讲其中一个最基本的特殊方法：__init__()，关于其他 Python 的魔法方法，接下来会专门用一个章节来详细讲解。

通常把__init__()方法称为构造方法，__init__()方法的魔力体现在只要实例化一个对象，这个方法就会在对象被创建时自动调用(在 C++里你也可以看到类似的东西，叫“构造函数”)。

其实,实例化对象时是可以传入参数的,这些参数会自动传入__init__()方法中,可以通过重写这个方法来自定义对象的初始化操作。举个例子:

```
>>> class Potato:
    def __init__(self, name):
        self.name = name
    def kick(self):
        print("我叫%s,噢~谁踢我?!" % self.name)

>>> p = Potato("土豆")
>>> p.kick()
我叫土豆,噢~谁踢我?!
```

### 11.3.3 公有和私有

一般面向对象的编程语言都会区分公有和私有的数据类型,像 C++ 和 Java 它们使用 public 和 private 关键字,用于声明数据是公有的还是私有的,但在 Python 中并没有用类似的关键字来修饰。

难道 Python 所有东西都是透明的？也不全是,默认上对象的属性和方法都是公开的,可以直接通过点操作符(.)进行访问:

```
>>> class Person:
    name = "小甲鱼"

>>> p = Person()
>>> p.name
'小甲鱼'
```

为了实现类似私有变量的特征,Python 内部采用了一种叫 name mangling(名字改编)的技术,在 Python 中定义私有变量只需要在变量名或函数名前加上"__"两个下划线,那么这个函数或变量就会成为私有的了:

```
>>> class Person:
    __name = "小甲鱼"

>>> p = Person()
>>> p.__name
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    p.__name
AttributeError: 'Person' object has no attribute '__name'
```

这样在外部将变量名"隐藏"起来了,理论上如果要访问,就需要从内部进行:

```
>>> class Person:
    def __init__(self, name):
        self.__name = name
    def getName(self):
        return self.__name
```

```
>>> p = Person("小甲鱼")
>>> p.__name
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    p.__name
AttributeError: 'Person' object has no attribute '__name'
>>> p.getName()
'小甲鱼'
```

但是你认真琢磨一下这个技术的名字 name mangling(名字改编),那就不难发现其实 Python 只是动了一下手脚,把双下横线开头的变量进行了改名而已。实际上在外部你使用“_类名__变量名”即可访问双下横线开头的私有变量了:

```
>>> p._Person__name
'小甲鱼'
```

(注:Python 目前的私有机制其实是伪私有,Python 的类是没有权限控制的,所有变量都是可以被外部调用的。最后的这部分有些读者(尤其是没有接触过面向对象编程的读者)可能看不懂,想不明白有什么用?没事,先放着,下节讲完继承机制你就会豁然开朗了。)

## 11.4　继承



现在需要扩展游戏,对鱼类进行细分,有金鱼(Goldfish)、鲤鱼(Carp)、三文鱼(Salmon),还有鲨鱼(Shark)。那么我们就再思考一个问题:能不能不要每次都从头到尾去重新定义一个新的鱼类呢?因为我们知道大部分鱼的属性和方法是相似的,如果有一种机制可以让这些相似的东西得以自动传递,那就方便快捷多了。没错,你猜到了,这种机制就是今天要讲的:继承。

语法很简单:

```
class 类名(被继承的类):
    …
```

被继承的类称为基类、父类或超类;继承者称为子类,一个子类可以继承它的父类的任何属性和方法。举个例子:

```
>>> class Parent:
    def hello(self):
        print("正在调用父类的方法…")

>>> class Child(Parent):
    pass

>>> p = Parent()
>>> p.hello()
正在调用父类的方法…
>>> c = Child()
>>> c.hello()
正在调用父类的方法…
```

需要注意的是，如果子类中定义与父类同名的方法或属性，则会自动覆盖父类对应的方法或属性：

```
>>> class Child(Parent):
    def hello(self):
        print("正在调用子类的方法…")

>>> c = Child()
>>> c.hello()
正在调用子类的方法…
```

好，那尝试来写一下刚才提到的金鱼(Goldfish)、鲤鱼(Carp)、三文鱼(Salmon)，还有鲨鱼(Shark)的例子：

```
# p11_2.py
import random as r

class Fish:
    def __init__(self):
        self.x = r.randint(0, 10)
        self.y = r.randint(0, 10)

    def move(self):
        # 这里主要演示类的继承机制,就不考虑检查场景边界和移动方向的问题

        # 假设所有鱼都是一路向西游
        self.x -= 1
        print("我的位置是: ", self.x, self.y)

class Goldfish(Fish):
    pass

class Carp(Fish):
    pass

class Salmon(Fish):
    pass

# 上边几个都是食物,食物不需要有个性,所以直接继承 Fish 类的全部属性和方法即可
# 下边定义鲨鱼类,这个是吃货,除了继承 Fish 类的属性和方法,还要添加一个吃的方法

class Shark(Fish):
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print("吃货的梦想就是天天有得吃^_^")
            self.hungry = False
        else:
            print("太撑了,吃不下!")

>>> # 先运行 p11-2.py
```

```
>>> fish = Fish()
>>> # 试试小鱼能不能移动
>>> fish.move()
我的位置是: 5 10
>>> goldfish = Goldfish()
>>> goldfish.move()
我的位置是: 9 10
>>> goldfish.move()
我的位置是: 8 10
>>> goldfish.move()
我的位置是: 7 10
>>> # 可见金鱼确实在一路向西…
>>> # 下边尝试生成鲨鱼
>>> shark = Shark()
>>> # 试试这货能不能吃东西?
>>> shark.eat()
吃货的梦想就是天天有得吃^_^
>>> shark.eat()
太撑了,吃不下!
>>> shark.move()
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    shark.move()
  File "E:\p11_2.py", line 13, in move
    self.x -= 1
AttributeError: 'Shark' object has no attribute 'x'
```

奇怪！同样是继承于 Fish 类，为什么金鱼(goldfish)可以移动，而鲨鱼(shark)一移动就报错呢？

其实这里抛出的异常说得很清楚了：Shark 对象没有 x 属性。原因其实是这样的：在 Shark 类中，重写了魔法方法__init__，但新的__init__方法里边没有初始化鲨鱼的 x 坐标和 y 坐标，因此调用 move 方法就会出错。那么解决这个问题的方案就很明显了，应该在鲨鱼类中重写__init__方法的时候先调用基类 Fish 的__init__方法。

下面介绍两种可以实现的技术：

- 调用未绑定的父类方法。
- 使用 super 函数。

## 11.4.1 调用未绑定的父类方法

调用未绑定的父类方法，听起来有些高深，但大家参考下面改写的代码就能心领神会了：

```
class Shark(Fish):
    def __init__(self):
        Fish.__init__(self)
        self.hungry = True
```

再运行下发现鲨鱼也可以成功移动了：

```
>>> # 先运行修改后的 p11-2.py
>>> shark = Shark()
```

```
>>> shark.move()
我的位置是: 7 9
>>> shark.move()
我的位置是: 6 9
```

这里需要注意的是这个 self 并不是父类 Fish 的实例对象，而是子类 Shark 的实例对象，所以这里说的未绑定是指并不需要绑定父类的实例对象，使用子类的实例对象代替即可。

有些读者可能不大理解，没关系，这一点都不重要！因为在 Python 中，有一个更好的方案可以取代它，就是使用 super 函数。

### 11.4.2 使用 super 函数

super 函数能够帮我自动找到基类的方法，而且还为我们传入了 self 参数，这样就不需要做这些事情了：

```
# 将 p11-2.py 鲨鱼的代码作如下修改
class Shark(Fish):
    def __init__(self):
        super().__init__()
        self.hungry = True
```

运行后得到同样的结果：

```
>>> # 先运行修改后的 p11-2.py
>>> shark = Shark()
>>> shark.move()
我的位置是: 6 1
>>> shark.move()
我的位置是: 5 1
```

super 函数的"超级"之处在于你不需要明确给出任何基类的名字，它会自动帮您找出所有基类以及对应的方法。由于你不用给出基类的名字，这就意味着如果需要改变类继承关系，只要改变 class 语句里的父类即可，而不必在大量代码中去修改所有被继承的方法。

## 11.5 多重继承

除此之外 Python 还支持多继承，就是可以同时继承多个父类的属性和方法：

```
class 类名(父类 1,父类 2,父类 3,…):
    …
```

```
>>> class Base1:
    def foo1(self):
        print("我是 foo1,我在 Base1 中…")

>>> class Base2:
    def foo2(self):
        print("我是 foo2,我在 Base2 中…")
```

```
>>> class C(Base1, Base2):
        pass

>>> c = C()
>>> c.foo1()
我是 foo1,我在 Base1 中…
>>> c.foo2()
我是 foo2,我在 Base2 中…
```

上面就是基本的多重继承语法。但多重继承其实很容易导致代码混乱,所以当你不确定是否真的必须使用多重继承的时候,请尽量避免使用它,因为有些时候会出现不可预见的 BUG。

【扩展阅读】多重继承的陷阱:钻石继承(菱形继承)问题(http://bbs.fishc.com/thread-48759-1-1.html)。

## 11.6 组合



前边先是学习了继承的概念,然后又学习了多重继承,但听到大牛们强调说不到必要的时候不使用多重继承。哎呀,这可让大家烦恼死了,就像上回我们有了乌龟类、鱼类,现在要求定义一个类,叫水池,水池里要有乌龟和鱼。用多重继承就显得很奇怪,因为水池和乌龟、鱼是不同物种,那要怎样才能把它们组合成一个水池的类呢?

在 Python 里其实很简单,直接把需要的类放进去实例化就可以了,这就叫组合:

```
# p11_3.py
class Turtle:
    def __init__(self, x):
        self.num = x

class Fish:
    def __init__(self, x):
        self.num = x

class Pool:
    def __init__(self, x, y):
        self.turtle = Turtle(x)
        self.fish = Fish(y)
    def print_num(self):
        print("水池里总共有乌龟 %d 只,小鱼 %d 条!" % (self.turtle.num, self.fish.num))

>>> # 先运行 p11_3.py
>>> pool = Pool(1, 10)
>>> pool.print_num()
水池里总共有乌龟 1 只,小鱼 10 条!
```

Python 的特性其实还支持另外一种很流行的编程模式:Mixin,有兴趣的朋友可以看看【扩展阅读】Mixin 编程机制(http://bbs.fishc.com/thread-48888-1-1.html)。

## 11.7 类、类对象和实例对象

先来分析一段代码：

```
>>> class C:
        count = 0

>>> a = C()
>>> b = C()
>>> c = C()
>>> print(a.count, b.count, c.count)
0 0 0
>>> c.count += 10
>>> print(a.count, b.count, c.count)
0 0 10
>>> C.count += 100
>>> print(a.count, b.count, c.count)
100 100 10
```

从上面的例子可以看出，对实例对象 c 的 count 属性进行赋值后，就相当于覆盖了类对象 C 的 count 属性。如图 11-1 所示，如果没有赋值覆盖，那么引用的是类对象的 count 属性。

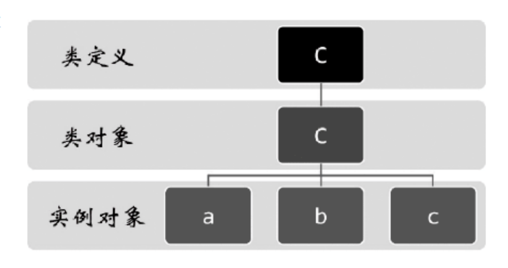



图 11-1　类、类对象和实例对象

需要注意的是，类中定义的属性是静态变量，也就是相当于 C 语言中加上 static 关键字声明的变量，类的属性是与类对象进行绑定，并不会依赖任何它的实例对象。这点待会儿继续讲解。

另外，如果属性的名字跟方法名相同，属性会覆盖方法：

```
class C:
    def x(self):
        print('Xman')

>>> c = C()
>>> c.x()
Xman
>>> c.x = 1
>>> c.x
1
>>> c.x()
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    c.x()
TypeError: 'int' object is not callable
```

为了避免名字上的冲突，大家应该遵守一些约定俗成的规矩：

- 类的定义要“少吃多餐”，不要试图在一个类里边定义出所有能想到的特性和方法，应该利用继承和组合机制来进行扩展。

- 用不同的词性命名，如属性名用名词、方法名用动词，并使用骆驼命名法[①]等。

## 11.8　到底什么是绑定

Python 严格要求方法需要有实例才能被调用，这种限制其实就是 Python 所谓的绑定概念。前面也粗略地解释了一下绑定，但有些读者可能会这么尝试，然后发现也可以调用：

```
>>> class BB:
    def printBB():
        print("no zuo no die")

>>> BB.printBB()
no zuo no die
```

但这样做会有一个问题，就是根据类实例化后的对象根本无法调用里边的函数：

```
>>> bb = BB()
>>> bb.printBB()
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    bb.printBB()
TypeError: printBB() takes 0 positional arguments but 1 was given
```

实际上由于 Python 的绑定机制，这里自动把 bb 对象作为第一个参数传入，所以才会出现 TypeError。

为了让大家更好地理解，再深入挖一挖：

```
>>> class CC:
    def setXY(self, x, y):
        self.x = x
        self.y = y
    def printXY(self):
        print(self.x, self.y)

>>> dd = CC()
```

可以使用__dict__查看对象所拥有的属性：

```
>>> dd.__dict__
{}
>>> CC.__dict__
mappingproxy({'__dict__': <attribute '__dict__' of 'CC' objects>, 'printXY': <function CC.printXY at 0x02D2D2B8>, '__weakref__': <attribute '__weakref__' of 'CC' objects>, 'setXY': <function CC.setXY at 0x02AC1420>, '__doc__': None, '__module__': '__main__'})
```

__dict__属性是由一个字典组成，字典中仅有实例对象的属性，不显示类属性和特殊属

① 骆驼式命名法(Camel-Case)又称驼峰命名法，是电脑程式编写时的一套命名规则(惯例)。正如它的名称 Camel Case 所表示的那样，是指混合使用大小写字母来构成变量和函数的名字，程序员们为了自己的代码能更容易在同行之间交流，所以多采取统一的可读性比较好的命名方式。

性，键表示的是属性名，值表示属性相应的数据值。

```
>>> dd.setXY(4, 5)
>>> dd.__dict__
{'x': 4, 'y': 5}
```

现在实例对象 dd 有了两个新属性，而且这两个属性仅属于实例对象的：

```
>>> CC.__dict__
mappingproxy({'__doc__': None, '__dict__': <attribute '__dict__' of 'CC' objects>, '__weakref__':
<attribute '__weakref__' of 'CC' objects>, 'printXY': <function CC.printXY at 0x0370D2B8>, '__
module__': '__main__', 'setXY': <function CC.setXY at 0x034A1420>})
```

为什么会这样呢？完全是归功于 self 参数：当实例对象 dd 去调用 setXY 方法的时候，它传入的第一个参数就是 dd，那么 self.x = 4，self.y = 5 也就相当于 dd.x = 4，dd.y = 5，所以你在实例对象，甚至类对象中都看不到 x 和 y，因为这两个属性是只属于实例对象 dd 的。

接着再深入一下，请思考：如果我把类实例删除掉，实例对象 dd 还能否调用 printXY 方法？

```
>>> del CC
    答案是可以的：
>>> dd.printXY()
4 5
```

## 11.9 一些相关的 BIF



下面介绍与类和对象相关的一些 BIF（内置函数）。

### 1. issubclass(class, classinfo)

如果第一个参数（class）是第二个参数（classinfo）的一个子类，则返回 True，否则返回 False：

（1）一个类被认为是其自身的子类。

（2）classinfo 可以是类对象组成的元组，只要 class 是其中任何一个候选类的子类，则返回 True。

（3）在其他情况下，会抛出一个 TypeError 异常。

```
>>> class A:
    pass

>>> class B(A):
    pass

>>> issubclass(B, A)
True
>>> issubclass(B, B)
True
>>> issubclass(B, object) # object 是所有类的基类
True
```

```
>>> class C:
    pass

>>> issubclass(B, C)
False
```

2．isinstance(object，classinfo)

如果第一个参数(object)是第二个参数(classinfo)的实例对象，则返回 True，否则返回 False：

(1) 如果 object 是 classinfo 的子类的一个实例，也符合条件。

(2) 如果第一个参数不是对象，则永远返回 False。

(3) classinfo 可以是类对象组成的元组，只要 object 是其中任何一个候选对象的实例，则返回 True。

(4) 如果第二个参数不是类或者由类对象组成的元组，会抛出一个 TypeError 异常。

```
>>> issubclass(B, C)
False
>>> b1 = B()
>>> isinstance(b1, B)
True
>>> isinstance(b1, C)
False
>>> isinstance(b1, A)
True
>>> isinstance(b1, (A, B, C))
True
```

Python 提供以下几个 BIF 用于访问对象的属性。

3．hasattr(object，name)

attr 即 attribute 的缩写，属性的意思。接下来将要介绍的几个 BIF 都是跟对象的属性有关系的，例如这个 hasattr()的作用就是测试一个对象里是否有指定的属性。

第一个参数(object)是对象，第二个参数(name)是属性名(属性的字符串名字)，举个例子：

```
class C:
    def __init__(self, x = 0):
        self.x = x

>>> c1 = C()
>>> hasattr(c1, 'x') # 注意,属性名要用引号括起来
True
```

4．getattr(object，name[，default])

返回对象指定的属性值，如果指定的属性不存在，则返回 default(可选参数)的值；若没有设置 default 参数，则抛出 ArttributeError 异常。

```
>>> getattr(c1, 'x')
0
>>> getattr(c1, 'y')
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    getattr(c1, 'y')
AttributeError: 'C' object has no attribute 'y'
>>> getattr(c1, 'y', '您所访问的属性不存在…')
'您所访问的属性不存在…'
```

### 5. setattr(object, name, value)

与 getattr()对应,setattr()可以设置对象中指定属性的值,如果指定的属性不存在,则会新建属性并赋值。

```
>>> setattr(c1, 'y', 'FishC')
>>> getattr(c1, 'y')
'FishC'
```

### 6. delattr(object, name)

与 setattr()相反,delattr()用于删除对象中指定的属性,如果属性不存在,则抛出 AttributeError 异常。

```
>>> delattr(c1, 'y')
>>> delattr(c1, 'z')
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    delattr(c1, 'z')
AttributeError: z
```

### 7. property(fget=None, fset=None, fdel=None, doc=None)

俗话说:条条大路通罗马。同样是完成一件事,Python 其实提供了好几个方式供你选择。property()是一个比较奇葩的 BIF,它的作用是通过属性来设置属性。说起来有点绕,看一下例子:

```
class C:
    def __init__(self, size=10):
        self.size = size

    def getSize(self):
        return self.size

    def setSize(self, value):
        self.size = value

    def delSize(self):
        del self.size
```

```
    x = property(getSize, setSize, delSize)

>>> c.x
10
>>> c.x = 12
>>> c.x
12
>>> c.size
12
>>> del c.x
>>> c.size
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    c.size
AttributeError: 'C' object has no attribute 'size'
```

property()返回一个可以设置属性的属性，当然如何设置属性还是需要人为来写代码。第一个参数是获得属性的方法名（例子中是 getSize），第二个参数是设置属性的方法名（例子中是 setSize），第三个参数是删除属性的方法名（例子中是 delSize）。

property()有什么作用呢？举个例子，在上面的例题中，为用户提供 setSize 方法名来设置 size 属性，并提供 getSize 方法名来获取属性。但是有一天你心血来潮，突然想对程序进行大改，就可能需要把 setSize 和 getSize 修改为 setXSize 和 getXSize，那就不得不修改用户调用的接口，这样的体验非常不好。

有了 property()，所有问题就迎刃而解了，因为像上边一样，为用户访问 size 属性只提供了 x 属性。无论内部怎么改动，只需要相应的修改 property()的参数，用户仍然只需要去操作 x 属性即可，没有任何影响。

很神奇是吧？想知道它是如何工作的？学完紧接着要讲的魔法方法，你就知道了。



# 第12章 魔法方法



## 12.1 构造和析构

在此之前,已经接触过 Python 最常用的魔法方法,小甲鱼也把魔法方法说得神乎其神,似乎用了就可以化腐朽为神奇,化干戈为玉帛,化不可能为可能!

说的这么厉害,那什么是魔法方法呢?

- 魔法方法总是被双下划线包围,例如__init__()。
- 魔法方法是面向对象的 Python 的一切,如果你不知道魔法方法,说明你还没能意识到面向对象的 Python 的强大。
- 魔法方法的"魔力"体现在它们总能够在适当的时候被调用。

### 12.1.1 __init__(self[, ...])

之前我们讨论过__init__()方法,说它相当于其他面向对象编程语言的构造方法,也就是类在实例化成对象的时候首先会调用的一个方法。

有读者可能会问:"有时候在类定义时写__init__()方法,有时候却没有,这是为什么呢?"这是我在论坛中看到的一个问题,我想应该不仅只有一位朋友有疑惑,所以在这里解释下:在现实生活中,有一种东西迫使我们去努力拼搏,使我们获得创造力和生产力,使我们不惜背井离乡来到一个陌生的城市承受孤独和寂寞,这个东西就叫需求……嗯,我想我已经很好地回答了这个问题。举个例子:

```
# p12_1.py
class Rectangle:
    """
    定义一个矩形类,
    需要长和宽两个参数,
    拥有计算周长和面积两个方法.
    需要对象在初始化的时候拥有"长"和"宽"两个参数,
    因此需要重写__init__()方法,因为我们说过,
    __init__()方法是类在实例化成对象的时候首先会调用的一个方法,
    大家可以理解吗?
    """
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def getPeri(self):
        return (self.x + self.y) * 2

    def getArea(self):
        return self.x * self.y

>>> # 先运行 p12_1.py
>>> rect = Rectangle(3, 4)
>>> rect.getPeri()
14
>>> rect.getArea()
12
```

这里需要注意的是，__init__()方法的返回值一定是 None，不能是其他：

```
>>> class A:
    def __init__(self):
        return "A for A - Cup"

>>> cup = A()
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    cup = A()
TypeError: __init__() should return None, not 'str'
```

所以一般在需要进行初始化的时候才重写__init__()方法，现在大家应该就可以理解造物者的逻辑了。但是你要知道，神之所以是神，是因为他做什么事都留有一手。其实，这个__init__()并不是实例化对象时第一个被调用的魔法方法。

### 12.1.2 __new__(cls[, ...])

__new__()才是在一个对象实例化的时候所调用的第一个方法。它跟其他魔法方法不同，它的第一个参数不是 self 而是这个类(cls)，而其他的参数会直接传递给__init__()方法的。

__new__()方法需要返回一个实例对象，通常是 cls 这个类实例化的对象，当然你也可以返回其他对象。

__new__()方法平时很少去重写它，一般让 Python 用默认的方案执行就可以了。但是有一种情况需要重写这个魔法方法，就是当继承一个不可变的类型的时候，它的特性就显得尤为重要了。

```
class CapStr(str):
    def __new__(cls, string):
        string = string.upper()
        return str.__new__(cls, string)

>>> a = CapStr("I love FishC.com")
>>> a
'I LOVE FISHC.COM'
```

这里返回 str.__new__(cls, string)这种做法是值得推崇的，只需要重写我们关注的那部分内容，然后其他的琐碎东西交给 Python 的默认机制去完成就可以了，毕竟它们出错的几率要比我们自己写小得多。

### 12.1.3 __del__(self)

如果说__init__()和__new__()方法是对象的构造器的话，那么 Python 也提供了一个析构器，叫作__del__()方法。当对象将要被销毁的时候，这个方法就会被调用。但一定要注意的是，并非 del x 就相当于自动调用 x.__del__()，__del__()方法是当垃圾回收机制回收这个对象的时候调用的。举个例子：

```
>>> class C:
	def __init__(self):
		print("我是__init__方法,我被调用了…")
	def __del__(self):
		print("我是__del__方法,我被调用了…")

>>> c1 = C()
我是__init__方法,我被调用了…
>>> c2 = c1
>>> c3 = c2
>>> del c1
>>> del c2
>>> del c3
我是__del__方法,我被调用了…
```

## 12.2 算术运算



现在来讲一个新的名词：工厂函数，不知道大家还有没有听过？其实在老早就一直在使用它，但由于那时候还没有学习类和对象，我知道那时候说了也是白说。但我知道现在来告诉大家，理解起来就不再是问题了。

Python2.2 以后，对类和类型进行了统一，做法就是将 int()、float()、str()、list()、tuple()这些 BIF 转换为工厂函数：

```
>>> type(len)
<class 'builtin_function_or_method'>
>>> type(int)
<class 'type'>
>>> type(dir)
<class 'builtin_function_or_method'>
>>> type(list)
<class 'type'>
```

看到没有，普通的 BIF 应该是<class 'builtin_function_or_method'>，而工厂函数则是<class 'type'>。大家有没有觉得这个<class 'type'>很眼熟，在哪里看过？没错啦，如果定义一个类：

```
>>> class C:
	pass

>>> type(C)
<class 'type'>
```

它的类型也是 type 类型,也就是类对象,其实所谓的工厂函数,其实就是一个类对象。当你调用它们的时候,事实上就是创建一个相应的实例对象:

```
>>> a = int('123')
>>> b = int('345')
>>> a + b
468
```

现在你是不是豁然发现:原来对象是可以进行计算的!其实你早该发现这个问题了,Python 中无处不对象,当在求 a+b 等于多少的时候,事实上 Python 就是在将两个对象进行相加操作。Python 的魔法方法还提供了自定义对象的数值处理,通过对下面这些魔法方法的重写,可以自定义任何对象间的算术运算。

## 12.2.1 算术操作符

表 12-1 列举了算数运算相关的魔法方法。

**表 12-1 算数运算相关的魔法方法**

| 魔 法 方 法 | 含 义 |
|---|---|
| __add__(self, other) | 定义加法的行为:+ |
| __sub__(self, other) | 定义减法的行为:- |
| __mul__(self, other) | 定义乘法的行为:* |
| __truediv__(self, other) | 定义真除法的行为:/ |
| __floordiv__(self, other) | 定义整数除法的行为://  |
| __mod__(self, other) | 定义取模算法的行为:% |
| __divmod__(self, other) | 定义当被 divmod()调用时的行为 |
| __pow__(self, other[, modulo]) | 定义当被 power()调用或 ** 运算时的行为 |
| __lshift__(self, other) | 定义按位左移位的行为:<< |
| __rshift__(self, other) | 定义按位右移位的行为:>> |
| __and__(self, other) | 定义按位与操作的行为:& |
| __xor__(self, other) | 定义按位异或操作的行为:^ |
| __or__(self, other) | 定义按位或操作的行为:| |

举个例子,下面定义一个比较特立独行的类:

```
>>> class New_int(int):
	def __add__(self, other):
		return int.__sub__(self, other)
	def __sub__(self, other):
		return int.__add__(self, other)

>>> a = New_int(3)
>>> b = New_int(5)
>>> a + b
```

```
-2
>>> a - b
8
```

那有些读者可能会问：我想自己写代码，不想通过调用 Python 默认的方案行不行？答案是肯定行，但要格外小心！

```
>>> class Try_int(int):
	def __add__(self, other):
		return self + other
	def __sub__(self, other):
		return self - other

>>> a = Try_int(1)
>>> b = Try_int(3)
>>> a + b
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    a + b
  File "<pyshell#6>", line 3, in __add__
    return self + other
  File "<pyshell#6>", line 3, in __add__
    return self + other
  # 此处省略很多行…
```

为什么会陷入无限递归呢？问题就出在这里：

```
def __add__(self, other):
		return self + other
```

当对象涉及加法操作时，自动调用魔法方法__add__()，但看看上边的魔法方法写的是什么？写的是 return self + other，也就是返回对象本身加另外一个对象，这不就又自动触发调用__add__()方法了吗？这样就形成了无限递归。所以，像下面这么写就不会触发无限递归了：

```
>>> class New_int(int):
	def __add__(self, other):
		return int(self) + int(other)
	def __sub__(self, other):
		return int(self) - int(other)

>>> a = New_int(1)
>>> b = New_int(3)
>>> a + b
4
```

上边介绍了很多有关算术运算的魔法方法，意思是当对象进行了相关的算术运算，自然而然就会自动触发对应的魔法方法。嘿，有悟性的读者就会说："哇，我似乎感觉到拥有了上帝的力量。没错吧？"

Python 正是如此，对于初学者，他们不知道魔法方法，所以默认的魔法方法会让他们以合乎逻辑的形式运行。但当你逐步深入学习，慢慢有了沉淀之后，你突然发现如果有更多的灵活性，就可以把程序写得更好……这时候，Python 也可以满足你。通过对指定魔法方法的重写，你完全可以让 Python 根据你的意愿去执行。

```
>>> class int(int):
        def __add__(self, other):
            return int.__sub__(self, other)

>>> a = int('5')
>>> b = int('3')
>>> a + b
2
```

当然，我这样做从逻辑上是说不过去的……我只是想跟大家说，随着学习的足够深入，Python 允许你做的事情就更多、更灵活！

## 12.2.2 反运算



表 12-2 列举了反运算相关的魔法方法。

**表 12-2 反运算相关的魔法方法**

| 魔法方法 | 含义 |
|---|---|
| __radd__(self, other) | 定义加法的行为：+（当左操作数不支持相应的操作时被调用） |
| __rsub__(self, other) | 定义减法的行为：-（当左操作数不支持相应的操作时被调用） |
| __rmul__(self, other) | 定义乘法的行为：*（当左操作数不支持相应的操作时被调用） |
| __rtruediv__(self, other) | 定义真除法的行为：/（当左操作数不支持相应的操作时被调用） |
| __rfloordiv__(self, other) | 定义整数除法的行为：//（当左操作数不支持相应的操作时被调用） |
| __rmod__(self, other) | 定义取模算法的行为：%（当左操作数不支持相应的操作时被调用） |
| __rdivmod__(self, other) | 定义当被 divmod()调用时的行为（当左操作数不支持相应的操作时被调用） |
| __rpow__(self, other) | 定义当被 power()调用或 ** 运算时的行为（当左操作数不支持相应的操作时被调用） |
| __rlshift__(self, other) | 定义按位左移位的行为：<<（当左操作数不支持相应的操作时被调用） |
| __rrshift__(self, other) | 定义按位右移位的行为：>>（当左操作数不支持相应的操作时被调用） |
| __rand__(self, other) | 定义按位与操作的行为：&（当左操作数不支持相应的操作时被调用） |
| __rxor__(self, other) | 定义按位异或操作的行为：^（当左操作数不支持相应的操作时被调用） |
| __ror__(self, other) | 定义按位或操作的行为：\|（当左操作数不支持相应的操作时被调用） |

不难发现，这里的反运算魔法方法跟上节介绍的算术运算符保持一一对应，不同之处就是反运算的魔法方法多了一个“r”，例如，__add__()就对应__radd__()。举个例子：

```
>>> a + b
# 这里加数是 a,被加数是 b,请问大家：这里是 a 主动还是 b 主动?
# 肯定是 a 主动,对不对?(就像"我请你吃饭"这句话,我肯定是主动,所以应该是由我给钱.但是,如果那天我刚好没带钱,那就叫蹭饭!但饭钱是一定要给的,那应该由谁来给?肯定就只能由 b 来给了.)
# 那反运算是同样一个道理,如果 a 对象的__add__()方法没有实现或者不支持相应的操作,那么 Python 就会自动调用 b 的__radd__()方法.
```

试一下：

```
>>> class Nint(int):
        def __radd__(self, other):
            return int.__sub__(other, self)
```

```
>>> a = Nint(5)
>>> b = Nint(3)
>>> a + b
8
# 由于a对象默认有__add__()方法,所以b的__radd__()没有执行
# 这样就有了:
>>> 1 + b
-2
```

关于反运算,这里还要注意一点:对于 a + b,b 的__radd__(self, other)的 self 是 b 对象,other 是 a 对象。

所以不能这么写:

```
>>> class Nint(int):
    def __rsub__(self, other):
        return int.__sub__(self, other)

>>> a = Nint(5)
>>> 3 - a
2
```

所以对于注重操作数顺序的运算符(例如减法、除法、移位),在重写反运算魔法方法的时候,就一定要注意顺序问题了。

### 12.2.3 增量赋值运算

Python 也有大量的魔术方法可以来定制增量赋值语句,增量赋值其实就是一种偷懒的形式,它将操作符与赋值来结合起来。例如:

```
>>> a = a + b
# 写成增量赋值的形式就是:
>>> a += b
```

### 12.2.4 一元操作符

一元操作符就是只有一个操作数的意思,像 a + b 这样,加号左右有 a、b 两个操作数,叫作二元操作符。只有一个操作数的,例如把减号放在一个操作数的前边,就是取这个操作数的相反数的意思,这时候管它叫负号。

Python 支持的一元操作符主要有__neg__()(表示正号行为),__pos__()(定义负号行为),__abs__()(定义当被 abs()调用时的行为,就是取绝对值的意思),还有一个__invert__()(定义按位取反的行为)。

## 12.3 简单定制



基本要求:

- 定制一个计时器的类。
- start 和 stop 方法代表启动计时和停止计时。

- 假设计时器对象 t1，print(t1)和直接调用 t1 均显示结果。
- 当计时器未启动或已经停止计时，调用 stop 方法会给予温馨的提示。
- 两个计时器对象可以进行相加：t1 + t2。
- 只能使用提供的有限资源完成。

这里需要限定你只能使用哪些资源，因为 Python 的模块是非常多的，你要是直接上网找个写好的模块进来，那就达不到锻炼的目的了。

下边是演示：

```
>>> t1 = MyTimer()
>>> t1
未开始计时!
>>> t1.stop()
提示: 请先调用 start() 开始计时!
>>> t1.start()
计时开始…
>>> t1
提示: 请先调用 stop() 结束计时!
>>> t1.stop()
计时结束!
>>> t1
总共运行了 5 秒
>>> t2 = MyTimer()
>>> t2.start()
计时开始…
>>> t2.stop()
计时结束!
>>> t2
总共运行了 6 秒
>>> t1 + t2
'总共运行了 11 秒'
```

你需要下面的资源：

- 使用 time 模块的 localtime 方法获取时间(有关 time 模块可参考：http://bbs.fishc.com/thread-51326-1-1.html)。
- time.localtime 返回 struct_time 的时间格式。
- 表现你的类：__str__()和__repr__()魔法方法。

```
>>> class A:
    def __str__(self):
        return "小甲鱼是帅哥"

>>> a = A()
>>> print(a)
小甲鱼是帅哥
>>> a
<__main__.A object at 0x03260F30>
>>> class B:
    def __repr__(self):
        return "小甲鱼是帅哥"
```

```
>>> b = B()
>>> b
小甲鱼是帅哥
```

有了这些知识,可以开始来编写代码了:

```
import time as t

class MyTimer:
    # 开始计时
    def start(self):
        self.start = t.localtime()
        print("计时开始…")
    # 停止计时
    def stop(slef):
        self.stop = t.localtime()
        print("计时结束!")
"""
```

好,万丈高楼平地起,把地基写好后,应该考虑怎么进行计算了.localtime()返回的是一个时间元组的结构,只需要前边6个元素,然后将stop的元素依次减去start对应的元素,将差值存放在一个新的列表里:

```
"""
    # 停止计时
    def stop(self):
        self.stop = t.localtime()
        self._calc()
        print("计时结束!")
    # 内部方法,计算运行时间
    def _calc(self):
        self.lasted = []
        self.prompt = "总共运行了"
        for index in range(6):
            self.lasted.append(self.stop[index] - self.start[index])
            self.prompt += str(self.lasted[index])
        print(self.prompt)

>>> t1 = MyTimer()
>>> t1.start()
计时开始…
>>> t1.stop()
总共运行了 000003
计时结束!
"""
```

已经基本实现计时功能了,接下来需要完成"print(t1)和直接调用t1均显示结果",那就要通过重写__str__()和__repr__()魔法方法来实现:

```
"""
    def __str__(self):
        return self.prompt
    __repr__ = __str__

>>> t1 = MyTimer()
>>> t1.start()
```

```
计时开始…
>>> t1.stop()
计时结束!
>>> t1
总共运行了 000002
```

似乎做得不错了，但这里还有一些问题。假设用户不按常理出牌，问题就会很多：

```
>>> t1 = MyTimer()
>>> t1
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    t1
  File "C:\Python34\lib\idlelib\rpc.py", line 614, in displayhook
    text = repr(value)
  File "C:\Users\FishC000\Desktop\test.py", line 5, in __str__
    return self.prompt
AttributeError: 'MyTimer' object has no attribute 'prompt'
```

当直接执行 t1 的时候，Python 会调用__str__()魔法方法，但它却说这个类没有 prompt 属性。prompt 属性在哪里定义的？在_calc()方法里定义的，对不？但是没有执行 stop()方法，_calc()方法就没有被调用到，所以也就没有 prompt 属性的定义了。

要解决这个问题也很简单，大家应该还记得在类里边，用得最多的一个魔法方法是什么？是__init__()嘛，所有属于实例对象的变量只要在这里边先定义，就不会出现这样的问题了。

```
…
    def __init__(self):
        self.prompt = "未开始计时!"
        self.lasted = []
        self.start = 0
        self.stop = 0
…
>>> t1 = MyTimer()
>>> t1
未开始计时!
>>> t1.start()
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    t1.start()
TypeError: 'int' object is not callable
```

这里又出错了(当然我是故意的)，大家先检查一下是什么问题？

其实会导致这个问题，是因犯了一个微妙的错误，这样的错误通常很容易疏忽，而且很难排查。Python 这里抛出了一个异常：TypeError：'int' object is not callable。

仔细瞧，在调用 start()方法的时候报错，也就是说，Python 认为 start 是一个整型变量，而不是一个方法。为什么呢？大家看__init__()方法里，是不是也命名了一个叫作 self.start 的变量，如果类中的方法名和属性同名，属性会覆盖方法。

好了，让我们把所有的 self.start 和 self.end 都改为 self.begin 和 self.end 吧！

现在程序没问题了，但显示时间是 000003 这样不大人性化，还是希望可以按照“年月日小时分钟秒”这么去显示，然后值为 0 的就不显示啦，这样才是人看的嘛，对不对?！所以这里添

加一个列表用来存放对应的单位：

```
...
    def __init__(self):
        self.unit = ['年', '月', '天', '小时', '分钟', '秒']
        self.prompt = "未开始计时!"
        self.lasted = []
        self.begin = 0
        self.end = 0
    # 计算运行时间
    def _calc(self):
        self.lasted = []
        self.prompt = "总共运行了"
        for index in range(6):
            self.lasted.append(self.end[index] - self.begin[index])
            if self.lasted[index]:
                self.prompt += (str(self.lasted[index]) + self.unit[index])
...
>>> t1 = MyTimer()
>>> t1.start()
计时开始...
>>> t1.stop()
计时结束!
>>> t1
总共运行了 2 秒
```

然后在适当的地方增加温馨提示：

```
...
    # 开始计时
    def start(self):
        self.begin = t.localtime()
        self.prompt = "提示: 请先调用 stop() 结束计时!"
        print("计时开始...")
    # 停止计时
    def stop(self):
        if not self.begin:
            print("提示: 请先调用 start() 开始计时!")
        else:
            self.end = t.localtime()
            self._calc()
            print("计时结束!")
    # 计算运行时间
    def _calc(self):
        self.lasted = []
        self.prompt = "总共运行了"
        for index in range(6):
            self.lasted.append(self.end[index] - self.begin[index])
            if self.lasted[index]:
                self.prompt += (str(self.lasted[index]) + self.unit[index])
        # 为下一轮计算初始化变量
        self.begin = 0
        self.end = 0
...
```

最后，再重写一个魔法方法__add__()，让两个计时器对象相加会自动返回时间的和：

```
…
    def __add__(self, other):
        prompt = "总共运行了"
        result = []
        for index in range(6):
            result.append(self.lasted[index] + other.lasted[index])
            if result[index]:
                prompt += (str(result[index]) + self.unit[index])
        return prompt
…
>>> t1 = MyTimer()
>>> t1
未开始计时!
>>> t1.stop()
提示：请先调用 start() 开始计时!
>>> t1.start()
计时开始…
>>> t1
提示：请先调用 stop() 结束计时!
>>> t1.stop()
计时结束!
>>> t1
总共运行了 8 秒
>>> t2 = MyTimer()
>>> t2.start()
计时开始…
>>> t2.stop()
计时结束!
>>> t2
总共运行了 4 秒
>>> t1 + t2
'总共运行了 12 秒'
```

看上去代码是不错，也能正常计算了。但是，这个程序有几点不足还需要大家课后来思考一下如何修改：

(1) 如果开始计时的时间是(2022 年 2 月 22 日 16:30:30)，停止时间是(2025 年 1 月 23 日 15:30:30)，那么按照用停止时间减开始时间的计算方式就会出现负数，你应该对此做一些转换。

(2) 现在的计算机速度都非常快，而这个程序最小的计算单位却只是秒，精度是远远不够的。

## 12.4 属性访问



通常可以通过点(.)操作符的形式去访问对象的属性，在 11.9 节中也谈到了如何通过几个 BIF 适当地去访问属性：

```
>>> class C:
```

```
    def __init__(self):
        self.x = 'X-man'

>>> c = C()
>>> c.x
'X-man'
>>> getattr(c, 'x', '木有这个属性')
'X-man'
>>> getattr(c, 'y', '木有这个属性')
'木有这个属性'
>>> setattr(c, 'y', 'Yellow')
>>> getattr(c, 'y', '木有这个属性')
'Yellow'
>>> delattr(c, 'x')
>>> c.x
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    c.x
AttributeError: 'C' object has no attribute 'x'
```

然后还介绍了一个叫作 property()函数的用法，这个 property()使得我们可以用属性去访问属性：

```
# p12_2.py
class C:
    def __init__(self, size=10):
        self.size = size
    def getSize(self):
        return self.size
    def setSize(self, value):
        self.size = value
    def delSize(self):
        del self.size
    x = property(getSize, setSize, delSize)

>>> # 先运行 p12_2.py
>>> c = C()
>>> c.x
10
>>> c.x = 12
>>> c.x
12
>>> c.size
12
>>> del c.x
>>> c.size
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    c.size
AttributeError: 'C' object has no attribute 'size'
```

那么关于属性访问，肯定也有相应的魔法方法来管理。通过对这些魔法方法的重写，你可

以随心所欲地控制对象的属性访问。大家是不是想想就有点小激动了呢？来吧，让我们开始吧！

表 12-3 列举了属性相关的魔法方法。

**表 12-3 属性相关的魔法方法**

| 魔 法 方 法 | 含 义 |
|---|---|
| __getattr__(self, name) | 定义当用户试图获取一个不存在的属性时的行为 |
| __getattribute__(self, name) | 定义当该类的属性被访问时的行为 |
| __setattr__(self, name, value) | 定义当一个属性被设置时的行为 |
| __delattr__(self, name) | 定义当一个属性被删除时的行为 |

做个小测试：

```
# p12_3.py
class C:
    def __getattribute__(self, name):
        print('getattribute')
        # 使用 super()调用 object 基类的__getattribute__()方法
        return super().__getattribute__(name)
    def __setattr__(self, name, value):
        print('setattr')
        super().__setattr__(name, value)
    def __delattr__(self, name):
        print('delattr')
        super().__delattr__(name)
    def __getattr__(self, name):
        print('getattr')
```

```
>>> # 先运行 p12_3.py
>>> c = C()
>>> c.x
getattribute
getattr
>>> c.x = 1
setattr
>>> c.x
getattribute
1
>>> del c.x
delattr
>>> setattr(c, 'y', 'Yellow')
setattr
```

这几个魔法方法在使用上需要注意的是，有一个死循环的陷阱，初学者比较容易中招，还是通过一个实例来讲解！写一个矩形类(Rectangle)，默认有宽(width)和高(height)两个属性；如果为一个叫 square 的属性赋值，那么说明这是一个正方形，值就是正方形的边长，此时宽和高都应该等于边长。

```
# p12_4.py
class Rectangle:
```

```
    def __init__(self, width = 0, height = 0):
        self.width = width
        self.height = height
    def __setattr__(self, name, value):
        if name == 'square':
            self.width = value
            self.height = value
        else:
            self.name = value
    def getArea(self):
        return self.width * self.height
```

```
>>> # 先运行 p12_4.py
>>> r1 = Rectangle(4, 5)
Traceback (most recent call last):
  File "<pyshell#181>", line 1, in <module>
    r1 = Rectangle(4, 5)
  File "E:\p12_4.py", line 3, in __init__
    self.width = width
  File " E:\p12_4.py", line 11, in __setattr__
self.name = value
  File " E:\p12_4.py", line 11, in __setattr__
self.name = value
  …
RuntimeError: maximum recursion depth exceeded while calling a Python object
```

这是为什么呢？

分析一下：实例化对象，调用__init__()方法，在这里给 self.width 和 self.heigth 分别初始化赋值。一发生赋值操作，就会自动触发__setattr__()魔法方法，width 和 height 两个属性被赋值，于是执行 else 的下边的语句，就又变成了 self.width=value，那么就相当于又触发了__setattr__()魔法方法了，死循环陷阱就是这么来的。

那怎么解决呢？我这里说两个方法。第一个就是跟刚才一样，用 super()来调用基类的__setattr__()，那么这样就依赖基类的方法来实现赋值：

```
…
else:
super().__setattr__(name, value)
…
```

```
>>> # 先执行修改后的 p12_4.py
>>> r1 = Rectangle(4, 5)
>>> r1.getArea()
20
>>> r1.square = 10
>>> r1.getArea()
100
```

另一种方法就是给特殊属性__dict__赋值。对象有一个特殊的属性，叫作__dict__，它的作用是以字典的形式显示出当前对象的所有属性以及相对应的值：

```
>>> r1.__dict__
```

```
{'height': 10, 'width': 10}
```

可以这么改:

```
…
else:
self.__dict__[name] = value
…
>>> # 先执行修改后的 p12_4.py
>>> r1 = Rectangle(4, 5)
>>> r1.getArea()
20
>>> r1.square = 10
>>> r1.getArea()
100
```

## 12.5　描述符(property 的原理)



此前提到过 property()函数,这不提不要紧,一提不得了,把大家的好奇心都给提起来了。大家都在问:“这 property()到底被下了什么药?怎么这么神奇?”如果你想知道 property()函数的实现原理,那么本节的内容就不能错过。

本节要讲的内容叫作描述符(descriptor),用一句话来解释,描述符就是将某种特殊类型的类的实例指派给另一个类的属性。那什么是特殊类型的类呢?就是至少要在这个类里边定义__get__()、__set__()或__delete__()三个特殊方法中的任意一个。

表 12-4 列举了描述符相关的魔法方法。

**表 12-4　描述符相关的魔法方法**

| 魔法方法 | 含　义 |
|---|---|
| __get__(self, instance, owner) | 用于访问属性,它返回属性的值 |
| __set__(self, instance, value) | 将在属性分配操作中调用,不返回任何内容 |
| __delete__(self, instance) | 控制删除操作,不返回任何内容 |

举个最直观的例子:

```
# p12_5.py
class MyDescriptor:
    def __get__(self, instance, owner):
        print("getting...", self, instance, owner)
    def __set__(self, instance, value):
        print("setting...", self, instance, value)
    def __delete__(self, instance):
        print("deleting...", self, instance)

class Test:
    x = MyDescriptor()
```

由于 MyDescriptor 实现了__get__()、__set__()和__delete__()方法,并且将它的类实例指派给 Test 类的属性,所以 MyDescriptor 就是所谓描述符类。到这里,大家有没有看到 property()的影子?

好，实例化 Test 类，然后尝试对 x 属性进行各种操作，看看描述符类会有怎样的响应：

```
>>> test = Test()
>>> test.x
getting... <__main__.myDescriptor object at 0x02D7FE90 > <__main__.Test object at 0x02FE0930 >
<class '__main__.Test'>
```

当访问 x 属性的时候，Python 会自动调用描述符的__get__()方法，几个参数的内容分别是：self 是描述符类自身的实例；instance 是这个描述符的拥有者所在的类的实例，在这里也就是 Test 类的实例；owner 是这个描述符的拥有者所在的类本身。

```
>>> test.x = 'X-man'
setting... <__main__.MyDescriptor object at 0x02D7FE90 > <__main__.Test object at 0x02FE0930
> X-man
```

对 x 属性进行赋值操作的时候，Python 会自动调用__set__()方法，前两个参数跟__get__()方法是一样的，最后一个参数 value 是等号右边的值。

最后一个 del 操作也是同样的道理：

```
>>> del test.x
deleting... <__main__.MyDescriptor object at 0x02D7FE90 > <__main__.Test object at 0x02FE0930 >
```

只要弄清楚描述符，那么 property 的秘密就不再是秘密了！property 事实上就是一个描述符类。下边就定义一个属于我们自己的 MyProperty：

```
# p12_6.py
class MyProperty:
    def __init__(self, fget = None, fset = None, fdel = None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
    def __get__(self, instance, owner):
        return self.fget(instance)
    def __set__(self, instance, value):
        self.fset(instance, value)
    def __delete__(self, instance):
        self.fdel(instance)

class C:
    def __init__(self):
        self._x = None
    def getX(self):
        return self._x
    def setX(self, value):
        self._x = value
    def delX(self):
        del self._x

    x = MyProperty(getX, setX, delX)
```

```
>>> # 先执行 p12_6.py
>>> c = C()
```

```
>>> c.x = 'X-man'
>>> c.x
'X-man'
>>> c._x
'X-man'
>>> del c.x
>>> c._x
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    c._x
AttributeError: 'C' object has no attribute '_x'
```

看，这不就自己实现property()函数了嘛，简单吧?!

最后讲一个实例：先定义一个温度类，然后定义两个描述符类用于描述摄氏度和华氏度两个属性。两个属性会自动进行转换，也就是说，你可以给摄氏度这个属性赋值，然后打印的华氏度属性是自动转换后的结果。

```
# p12_7.py
class Celsius:
    def __init__(self, value=26.0):
        self.value = float(value)
    def __get__(self, instance, owner):
        return self.value
    def __set__(self, instance, value):
        self.value = float(value)

class Fahrenheit:
    def __get__(self, instance, owner):
        return instance.cel * 1.8 + 32
    def __set__(self, instance, value):
        instance.cel = (float(value) - 32) / 1.8

class Temperature:
    cel = Celsius()
    fah = Fahrenheit()

>>> # 先执行 p12_7.py
>>> temp = Temperature()
>>> temp.cel
26.0
>>> temp.fah
78.80000000000001
```

## 12.6 定制序列



常言道，无规矩不成方圆，讲的是万事万物的发展都是要在一定的规则下进行，只有遵照一定的协议去做了，事情才能往正确的方向上发展。

本节要谈的是定制容器，要想成功地实现容器的定制，便需要先谈一谈协议。协议是什么

呢？协议(Protocol)与其他编程语言中的接口很相似，它规定哪些方法必须要定义。然而，在Python中的协议就显得不那么正式。事实上，在Python中，协议更像是一种指南。

这有点像Python极力推崇的鸭子类型(扩展阅读：http://bbs.fishc.com/thread-51471-1-1.html)，当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。Python就是这样，并不会严格地要求你一定要怎样去做，而是让你靠着自觉和经验把事情做好！

在Python中，像序列类型(如列表、元组、字符串)或映射类型(如字典)都是属于容器类型。本节来讲定制容器，那就必须要知道，定制容器有关的一些协议：

- 如果说你希望定制的容器是不可变的话，你只需要定义__len__()和__getitem__()方法。
- 如果你希望定制的容器是可变的话，除了__len__()和__getitem__()方法，你还需要定义__setitem__()和__delitem__()两个方法。

表12-5列举了定制容器类型相关的魔法方法及含义。

**表12-5　定制容器类型相关的魔法方法**

| 魔法方法 | 含　义 |
| --- | --- |
| __len__(self) | 定义当被len()函数调用时的行为(返回容器中元素的个数) |
| __getitem__(self, key) | 定义获取容器中指定元素的行为，相当于self[key] |
| __setitem__(self, key, value) | 定义设置容器中指定元素的行为，相当于self[key] = value |
| __delitem__(self, key) | 定义删除容器中指定元素的行为，相当于del self[key] |
| __iter__(self) | 定义当迭代容器中的元素的行为 |
| __reversed__(self) | 定义当被reversed()函数调用时的行为 |
| __contains__(self, item) | 定义当使用成员测试运算符(in或not in)时的行为 |

验证大家学习能力的时候到了。现在动动手，编写一个不可改变的自定义列表，要求记录列表中每个元素被访问的次数。

```
# p12_8.py
class CountList:
    def __init__(self, *args):
        self.values = [x for x in args]
        self.count = {}.fromkeys(range(len(self.values)), 0)
        # 这里使用列表的下标作为字典的键,注意不能用元素作为字典的键
        # 因为列表的不同下标可能有值一样的元素,但字典不能有两个相同的键
    def __len__(self):
        return len(self.values)
    def __getitem__(self, key):
        self.count[key] += 1
        return self.values[key]

>>> # 先运行 p12_8.py
>>> c1 = CountList(1, 3, 5, 7, 9)
>>> c2 = CountList(2, 4, 6, 8, 10)
>>> c1[1]
3
>>> c2[1]
4
```

```
>>> c1[1] + c2[1]
7
>>> c1.count
{0: 0, 1: 2, 2: 0, 3: 0, 4: 0}
>>> c2.count
{0: 0, 1: 2, 2: 0, 3: 0, 4: 0}
```

## 12.7　迭代器



自始至终,有一个概念一直在用,但我们却从来没有认真地去深入剖析它。这个概念就是迭代。迭代这个词听得很多了,现在不仅在数学领域使用这个词,我们经常听到类似这个产品经过多次迭代,质量和品质已经有了大幅度提高,这次事件纯属意外……

大家应该听出来了,迭代的意思类似于循环,每一次重复的过程被称为一次迭代的过程,而每一次迭代得到的结果会被用来作为下一次迭代的初始值。提供迭代方法的容器称为迭代器,通常接触的迭代器有序列(列表、元组、字符串)还有字典也是迭代器,都支持迭代的操作。

举个例子,通常使用 for 语句来进行迭代:

```
>>> for i in "FishC":
	print(i)
F
i
s
h
C
```

字符串就是一个容器,同时也是一个迭代器,for 语句的作用就是触发这个迭代器的迭代功能,每次从容器里依次拿出一个数据,这就是迭代操作。

字典和文件也是支持迭代操作的:

```
>>> links = {'鱼 C 工作室':'http://www.fishc.com', \
	 '鱼 C 论坛':'http://bbs.fishc.com', \
	 '鱼 C 博客':'http://blog.fishc.com', \
	 '支持小甲鱼':'http://fishc.taobao.com'}
>>> for each in links:
	print('%s -> %s' % (each, links[each]))

鱼 C 博客 -> http://blog.fishc.com
鱼 C 论坛 -> http://bbs.fishc.com
鱼 C 工作室 -> http://www.fishc.com
支持小甲鱼 -> http://fishc.taobao.com
```

关于迭代,Python 提供了两个 BIF:

- iter()。
- next()。

对一个容器对象调用 iter()就得到它的迭代器,调用 next()迭代器就会返回下一个值,然后怎么样结束呢?如果迭代器没有值可以返回了,Python 会抛出一个叫作 StopIteration 的异常:

```
>>> string = "FishC"
>>> it = iter(string)
>>> next(it)
'F'
>>> next(it)
'i'
>>> next(it)
's'
>>> next(it)
'h'
>>> next(it)
'C'
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    next(it)
StopIteration
```

所以，利用这两个 BIF，可以分析出 for 语句其实是这么工作的：

```
>>> string = "FishC"
>>> it = iter(string)
>>> while True:
    try:
        each = next(it)
    except StopIteration:
        break
    print(each)
F
i
s
h
C
```

那么关于实现迭代器的魔法方法有两个：

- __iter__()。
- __next__()。

一个容器如果是迭代器，那就必须实现__iter__()魔法方法，这个方法实际上就是返回迭代器本身。接下来重点要实现的是__next__()魔法方法，因为它决定了迭代的规则。简单举个例子大家就清楚了：

```
>>> class Fibs:
    def __init__(self):
        self.a = 0
        self.b = 1
    def __iter__(self):
        return self
    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        return self.a
```

```
>>> fibs = Fibs()
>>> for each in fibs:
    if each < 20:
        print(each)
    else:
        break
1
1
2
3
5
8
13
```

好了，这个迭代器的唯一亮点就是没有终点，所以如果没有跳出循环，它会不断迭代下去。那可不可以加一个参数，用于控制迭代的范围呢？

```
>>> class Fibs:
    def __init__(self, n = 20):
        self.a = 0
        self.b = 1
        self.n = n
    def __iter__(self):
        return self
    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        if self.a > self.n:
            raise StopIteration
        return self.a

>>> fibs = Fibs()
>>> for each in fibs:
    print(each)
1
1
2
3
5
8
13

>>> fibs = Fibs(10)
>>> for each in fibs:
    print(each)
1
1
2
3
5
8
```

是不是很容易呢？嗯，Python 就是可以这么简简单单的一门语言！

## 12.8 生成器(乱入)



由于前边介绍了迭代器,所以这里趁热打铁,给大家讲一讲这个生成器。虽然说生成器和迭代器可以说是 Python 近几年来引入的最强大的两个特性,但是生成器的学习,并不涉及魔法方法,甚至它巧妙地避开了类和对象,仅通过普通的函数就可以实现了。

由于生成器的概念要比较高级一些,所以在函数章节就没有提及它,还是那句老话,因为那时候讲了也是白讲。学习就是这么一个渐进的过程,像上节介绍的迭代器,很多人学完之后感叹:哎呀,Python 怎么就这么简单呐!但如果在讲循环那个章节来讲迭代器的实现原理,那大家势必就会一头雾水了。

正如刚才说的,生成器其实是迭代器的一种实现,那既然迭代可以实现,为何还要生成器呢?有一句话叫“存在即合理”,生成器的发明一方面是为了使得 Python 更为简洁,因为,迭代器需要我们自己去定义一个类和实现相关的方法,而生成器则只需要在普通的函数中加上一个 yield 语句即可。

在另一个更重要的方面,生成器的发明,使得 Python 模仿协同程序的概念得以实现。所谓协同程序,就是可以运行的独立函数调用,函数可以暂停或者挂起,并在需要的时候从程序离开的地方继续或者重新开始。

对于调用一个普通的 Python 函数,一般是从函数的第一行代码开始执行,结束于 return 语句、异常或者函数所有语句执行完毕。一旦函数将控制权交还给调用者,就意味着全部结束。函数中做的所有工作以及保存在局部变量中的数据都将丢失。再次调用这个函数时,一切都将从头创建。

Python 是通过生成器来实现类似于协同程序的概念:生成器可以暂时挂起函数,并保留函数的局部变量等数据,然后在再次调用它的时候,从上次暂停的位置继续执行下去。

好,多说不如实干,举个例子:

```
>>> def myGen():
	print("生成器被执行!")
	yield 1
	yield 2

>>> myG = myGen()
>>> next(myG)
生成器被执行!
1
>>> next(myG)
2
>>> next(myG)
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    next(myG)
StopIteration
```

正如大家所看到的,当函数结束时,一个 StopIteration 异常就会被抛出。由于 Python 的 for 循环会自动调用 next()方法和处理 StopIteration 异常,所以 for 循环当然也是可以对生成

器产生作用的：

```
>>> for i in myGen():
    print(i)
生成器被执行!
1
2
```

像前面介绍的斐波那契的例子，也可以用生成器来实现：

```
>>> def fibs():
    a = 0
    b = 1
    while True:
        a, b = b, a + b
        yield a

>>> for each in fibs():
    if each > 100:
        break
    print(each)
1
1
2
3
5
8
13
21
34
55
89
```

事到如今，你应该已经很好地掌握了列表推导式，那大家猜猜看下边这个列表推导式表达的是啥意思：

```
>>> a = [i for i in range(100) if not(i % 2) and i % 3]
```

其实上边这个列表推导式求的就是100以内，能够被2整除，但不能够被3整除的所有整数：

```
>>> a
[2, 4, 8, 10, 14, 16, 20, 22, 26, 28, 32, 34, 38, 40, 44, 46, 50, 52, 56, 58, 62, 64, 68, 70, 74,
76, 80, 82, 86, 88, 92, 94, 98]
```

Python3 除了有列表推导式之外，还有字典推导式：

```
>>> b = {i:i % 2 == 0 for i in range(10)}
>>> b
{0: True, 1: False, 2: True, 3: False, 4: True, 5: False, 6: True, 7: False, 8: True, 9: False}
```

还有集合推导式：

```
>>> c = {i for i in [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 7, 7, 8]}
```

```
>>> c
{1, 2, 3, 4, 5, 6, 7, 8}
```

那这时有读者可能就会想："那按照这种剧情发展下去，应该会有字符串推导式和元组推导式吧？"不妨试试：

```
>>> d = "i for i in 'I love FishC.com!'"
>>> d
"i for i in 'I love FishC.com!'"
```

噢，不行，因为只要在双引号内，所有的东西都变成了字符串，所以不存在字符串推导式了。那元组推导式呢？

```
>>> e = (i for i in range(10))
>>> e
<generator object <genexpr> at 0x03135300>
```

咦？似乎这个不是什么推导式，大家看出来什么门道了吗？generator，多么熟悉的单词啊，不就是生成器嘛！没错，用普通的小括号括起来的正是生成器推导式，来证明一下：

```
>>> next(e)
0
>>> next(e)
1
>>> next(e)
2
>>> next(e)
3
>>> next(e)
4
```

用 for 语句把剩下的都打印出来：

```
>>> for each in e:
	print(each)
5
6
7
8
9
```

还有一点特性更牛，生成器推导式如果作为函数的参数，可以直接写推导式，而不用加小括号：

```
>>> sum(i for i in range(100) if i % 2)
2500
```

关于生成器的技术要点，这里小甲鱼还给大家转了一篇不错的文章，大家课后可以参考学习一下：解释 yield 和 Generators(生成器)(http://bbs.fishc.com/thread-56023-1-1.html)。

# 第13章 模块

## 13.1 模块就是程序



本节将给大家介绍一个新的知识，叫作模块。一早我们就说过模块是更高级的封装。说到封装，先回顾一下学过的有哪些？

容器，例如列表、元组、字符串、字典等，这些是对数据的封装。

函数，是对语句的封装。

类，是对方法和属性的封装，也就是对函数和数据的封装。

那本节学习的模块，又是怎样一种封装形式呢？要解答什么是模块这个问题，其实只需要用一句话就可以概括：模块就是程序。没错，模块，就是平时写的任何代码，保存的每一个.py结尾的文件，都是一个独立的模块。

举个简单的例子，在Python的安装目录下创建一个叫hello.py的文件，代码如下：

```
def hi():
    print("Hi everyone, I love FishC.com!")
```

当我把这个文件保存起来的时候，它就是一个独立的Python模块了(注意：为了让默认的IDLE可以找到这个模块，需要把文件放在Python的安装目录下)。

这时就可以在IDLE中导入模块了：

```
# 模块的名字就是刚刚保存的那个文件名(不带后缀哦)
>>> import hello
```

好，那试试调用一下hello模块中的hi函数：

```
>>> hi()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    hi()
NameError: name 'hi' is not defined
```

噢？出错了！从这个错误信息可以看出错误的根源是Python找不到hi()这个函数。为什么会这样呢？明明在hello文件中已经定义了hi()函数，这里Python却说我们未定义？

## 13.2 命名空间

什么是命名空间呢？命名空间(Namespace)表示标识符(identifier)的可见范围。一个标识符可在多个命名空间中定义，它在不同命名空间中的含义是互不相干的。

比如你们班里有个叫小花的同学，隔壁班也恰好有个叫小花的同学，由于她们在两个不同的班级，所以老师上课点名直接叫小花是没有问题的。但如果是期末统考，那么整个年级的成绩排名就分不清到底是你班还是隔壁班的小花排第一了。那怎么办呢？解决的方法很简单，就是在名字的前边写上相应的班级就可以了。在这个例子中，班级就是命名空间。

在 Python 中，每个模块都会维护一个独立的命名空间，我们应该将模块名加上，才能够正常使用模块中的函数：

```
>>> hello.hi()
Hi everyone, I love FishC.com!
```

## 13.3 导入模块

下面介绍一下几种导入模块的方法。

### 1. import 模块名

直接 import，但是在调用模块中的函数的时候，需要加上模块的命名空间。重新写一个例子，用于计算摄氏度和华氏度的相互转换：

```
# p13_1.py
def c2f(cel):
    fah = cel * 1.8 + 32
    return fah

def f2c(fah):
    cel = (fah - 32) / 1.8
    return cel
```

再写一个文件来导入刚才的模块：

```
# p13_2.py
import p13_1

print("32 摄氏度 = %.2f 华氏度" % p13_1.c2f(32))
print("99 华氏度 = %.2f 摄氏度" % p13_1.f2c(99))
```

### 2. from 模块名 import 函数名

刚才那种方法有些读者可能不是很喜欢，因为这个模块的名字太长了，每次调用模块里的函数都要写这么长的命名空间，真是费力不讨好又容易出错。所以呢，就有了这种方法。

这种导入方法会直接将模块的命名空间覆盖进来，所以调用的时候也就不需要再加上命

名空间了：

```
# p13_3.py
from p13_1 import c2f, f2c

print("32 摄氏度 = %.2f 华氏度" % c2f(32))
print("99 华氏度 = %.2f 摄氏度" % f2c(99))
```

这里还可以使用通配符星号(*)来导入模块中所有的命名空间：

```
from p13_1 import *
```

但是强烈要求大家不要使用这种方法，因为这样做会使得命名空间的优势荡然无存，一不小心还会陷入名字混乱的局面。

### 3. import 模块名 as 新名字

最后一种方法是作者本人大力推崇的，你可以用这种方法给导入的命名空间替换一个新的名字。

```
# p13_4.py
import p13_1 as tc

print("32 摄氏度 = %.2f 华氏度" % tc.c2f(32))
print("99 华氏度 = %.2f 摄氏度" % tc.f2c(99))
```

## 13.4 __name__ = '__main__'



前边已经介绍了模块的作用以及模块的用法。来回顾一下，模块的主要作用有哪些？

第一点无疑就是封装组织 Python 的代码，你想想，当代码量非常大的时候，可以有组织有纪律地根据不同的功能，将代码分割成不同的模块。这样，每个模块相互之间是独立开的。那大家说说，这代码是分开了容易阅读和测试，还是搭在一块容易？我们肯定是更愿意去阅读和测试一小段代码，而不是每一次都劈头盖脸地将一个程序从头读起。

然后，模块的另一个重要的特性就是实现代码的重用。比如你写了一段发送邮件的代码，多次优化之后发现这非常棒，你就可以封装成一个独立的模块，以后在任何程序需要发送邮件的时候，只需要导入这个模块就可以直接使用了，而不用在每个需要发送邮件的程序中都重复写同样的代码。

相信很多读者朋友已经开始去阅读别人的代码(注：通常通过阅读比你牛的人写的代码，会让你的技术水平飞速提高)，在阅读代码时，会发现很多代码中都有 if __name__ == '__main__'这么一行语句，但却不知道有什么用？

先举个例子，一般写完代码要先测试下：

```
# p13_5.py
def c2f(cel):
    fah = cel * 1.8 + 32
    return fah
```

```
def f2c(fah):
    cel = (fah - 32) / 1.8
    return cel

def test():
    print("测试,0 摄氏度 = %.2f 华氏度" % c2f(0))
    print("测试,0 华氏度 = %.2f 摄氏度" % f2c(0))

test()
```

单独这个运行是没问题的：

```
>>>
测试,0 摄氏度 = 32.00 华氏度
测试,0 华氏度 = -17.78 摄氏度
>>>
```

但如果是在另一个文件中(p13_6.py)导入后再调用：

```
# p13_6.py
import p13_5 as tc

print("32 摄氏度 = %.2f 华氏度" % tc.c2f(32))
print("99 华氏度 = %.2f 摄氏度" % tc.f2c(99))
```

就会出现问题：

```
>>>
测试,0 摄氏度 = 32.00 华氏度
测试,0 华氏度 = -17.78 摄氏度
32 摄氏度 = 89.60 华氏度
99 华氏度 = 37.22 摄氏度
>>>
```

Python 把模块中(p13_5.py)的测试函数也一块儿执行了，而这并不是我们想要的……避免这种情况的关键在于：让 Python 知道该模块是作为程序运行还是导入到其他程序中。为了实现这一点，需要使用模块的__name__属性：

```
>>> __name__
'__main__'
>>> tc.__name__
'p13_5'
```

在作为程序运行的时候，__name__属性的值是'__main__'，而作为模块导入的时候，这个值就是该模块的名字了。因此，你就不难理解 if __name__ == '__main__'这句代码的意思了。

```
# p13_7.py
def c2f(cel):
    fah = cel * 1.8 + 32
    return fah

def f2c(fah):
    cel = (fah - 32) / 1.8
```

```
    return cel

def test():
    print("测试,0 摄氏度 = %.2f 华氏度" % c2f(0))
    print("测试,0 华氏度 = %.2f 摄氏度" % f2c(0))

if __name__ == '__main__':
    test()
```

上面的代码确保只有单独运行 p13_7.py 时才会执行 test()函数。

## 13.5 搜索路径

现在遇到一个问题,写好的模块应该放在哪里?有读者可能会说:“不是应该放在和导入这个模块文件的源代码同一个文件夹内吗?”没错,这是一种方案。但有的读者可能不希望把所有的代码都放在一个框里,因为我想通过文件夹的方式更好地组织我的代码。可以做到吗?没问题,但在此之前你必须先理解搜索路径这个概念。

Python 模块的导入需要一个路径搜索的过程。就是说,你导入一个叫作 hello 的模块,那么 Python 会在预定义好的搜索路径中寻找一个叫作 hello.py 的模块文件——如果有,则导入模块;如果没有,则导入失败。而这个搜索路径,就是一组目录,可以通过 sys 模块中的 path 变量显示出来(不同的机器上显示的路径信息可能不一样):

```
>>> import sys
>>> sys.path
['', 'C:\\Python34\\Lib\\idlelib', 'C:\\WINDOWS\\SYSTEM32\\python34.zip', 'C:\\Python34\\DLLs',
'C:\\Python34\\lib', 'C:\\Python34', 'C:\\Python34\\lib\\site-packages']
```

列出的这些路径都是 Python 在导入模块操作时会去搜索的,尽管这些模块都可以使用,但 site-packages 目录是最佳的选择,因为它就是用来做这些事情的。

当然按照这个逻辑来说,只需要告诉 Python 你的模块文件在哪里找,Python 在导入模块的时候就能正确地找到它:

```
>>> # 假如存放模块(p13_7.py)的位置是: C:\Python34\test\M1
>>> import p13_7
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    import p13_7
ImportError: No module named 'p13_7'
>>> # 直接导入会出错,因为搜索路径并不包含模块所在的位置
>>> # 那把模块所在的位置添加到搜索路径中:
>>> sys.path.append("C:\\Python34\\test\\M1")
>>> sys.path
['', 'C:\\Python34\\Lib\\idlelib', 'C:\\WINDOWS\\SYSTEM32\\python34.zip', 'C:\\Python34\\DLLs',
'C:\\Python34\\lib', 'C:\\Python34', 'C:\\Python34\\lib\\site-packages', 'C:\\Python34\\test\\
M1']
>>> import p13_7 as tc
>>> print("32 摄氏度 = %.2f 华氏度" % tc.c2f(32))
32 摄氏度 = 89.60 华氏度
```

## 13.6 包

在实际的开发中，一个大型的系统有成千上万的 Python 模块是很正常的事情。单单用模块来定义 Python 的功能显然还不够，如果都放在一起显然不好管理并且有命名冲突的可能，因此 Python 中也出现了包的概念。

什么是包呢？事实上有点像刚刚所做的：把模块分门别类地存放在不同的文件夹，然后把各个文件夹的位置告诉 Python。只是包的实现要更为简洁一些。创建一个包的具体操作如下：

(1) 创建一个文件夹，用于存放相关的模块，文件夹的名字即包的名字；

(2) 在文件夹中创建一个__init__.py 的模块文件，内容可以为空；

(3) 将相关的模块放入文件夹中。

**注意**

> 注意第(2)步，必须要在每一个包目录下建立一个__init__.py 的模块，可以是一个空文件，也可以写一些初始化代码。这个是 Python 的规定，用来告诉 Python 将该目录当成一个包来处理。

接下来就是在程序中导入包的模块(包名.模块名)：

```
# p13_8.py
# 将 p13_7.py 放在了文件夹 M1 中
import M1.p13_7 as tc

print("32 摄氏度 = %.2f 华氏度" % tc.c2f(32))
print("99 华氏度 = %.2f 摄氏度" % tc.f2c(99))
```

看，程序正常执行：

```
>>>
32 摄氏度 = 89.60 华氏度
99 华氏度 = 37.22 摄氏度
>>>
```

## 13.7 像个极客一样去思考



Python 社区有句俗语叫“Python 自己带着电池”。什么意思呢？这要从 Python 的设计哲学说起……

Python 的设计哲学是“优雅、明确、简单”，因此，Python 开发者的哲学是“用一种方法，最好是只有一种方法来做一件事”。虽然作者常常鼓励大家多思考，条条大路通罗马，那是为了训练大家的发散性思维。但在正式编程中，如果有完善的并且经过严密测试过的模块可以实现，那么建议大家最好使用现成的模块。

随 Python 附带安装有 Python 标准库，说“Python 自己带着电池”，指的就是标准库里的

模块。这些模块都极其有用,一般常见的任务都有相应的模块可以实现。不过 Python 标准库里包含的模块有数百个之多,一个个模块单独来讲,那着实不现实。所以本节主要是将告诉大家如何独立地探究模块。

对于 Python 来说,学习资料其实一直都在身边。这里给大家分析下遇到问题,自己应该如何去找答案(其实 90%的问题都可以自己找到解决方法)。首先要找的就是 Python 的文档,选择 Help→Python Docs 选项。

来看下 Python 的官方帮助文档由几部分构成,如图 13-1 所示。

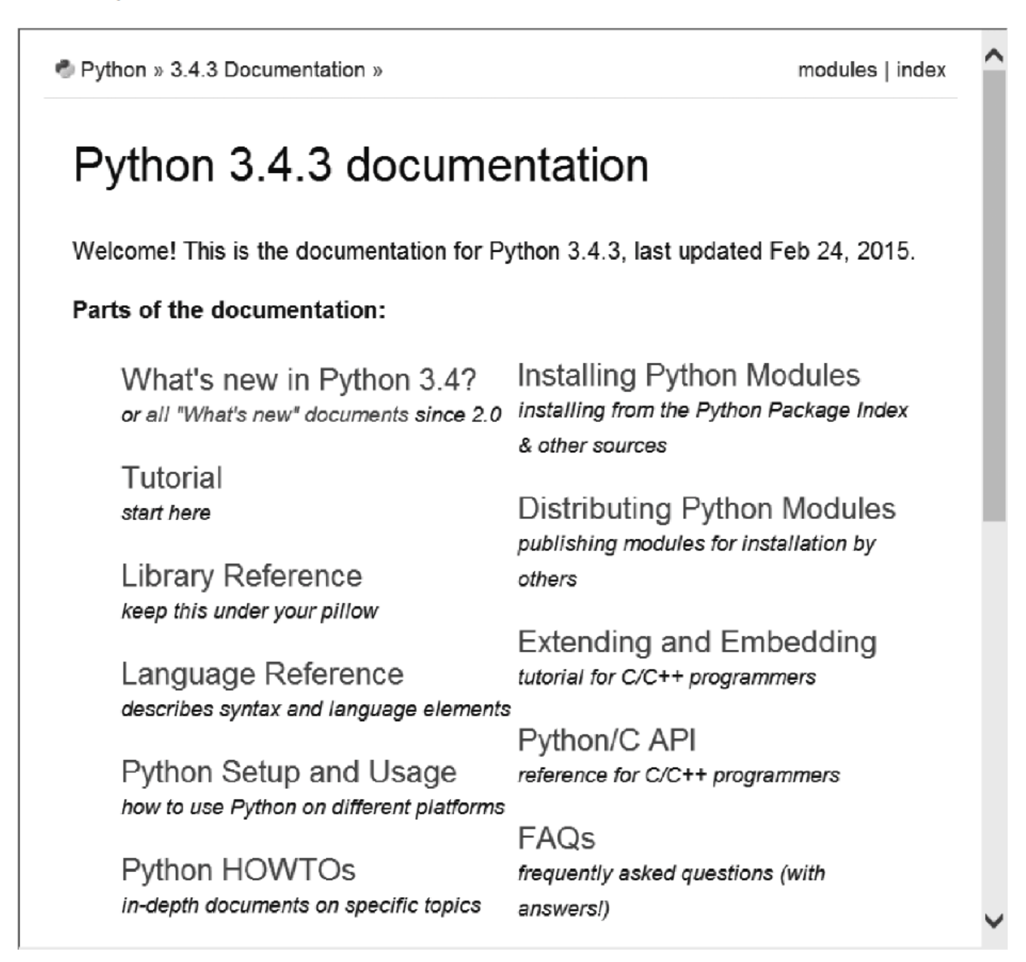



图 13-1 Python 官方帮助文档

Parts of the documentation:
Python 文档的主要组成部分

What's new in Python 3.4?
or all "What's new" documents since 2.0:
Python 3.4 有什么新的特性和改进?或者列举自 2.0 以后的所有新特性。

Tutorial:
简易教程,简单地介绍 Python 的语法,这个系列教程比它要详细得多,所以这里大家可以不

用看。

Library Reference：
Python 官方的枕边书，这里边详细地列举了 Python 所有的内置函数和标准库的各个模块的用法，非常详细，但是你看不完的，当作字典来查就可以了。
Installing Python Modules：
教你如何安装 Python 的第三方模块。

Distributing Python Modules：
教你如何发布 Python 的第三方模块。

Python 除了标准库的几百个模块之外，还有个 Pypi 社区，收集了全球的 Python 爱好者贡献的模块，你自己写了一个模块觉得要分享给世界，你也可以发布上去。

Language Reference：
讨论 Python 的语法和设计哲学。

Python Setup and Usage：
介绍在各个平台上如何使用 Python。

Python HOWTOs：
这里是深入探讨一些特定的主题。

Extending and Embedding：
介绍如何用 C 和 C++开发 Python 的扩展模块。

FAQs：
常见问题解答。

另外值得一提的是 PEP(如果查看文档经常会看到 PEP 后边加上一些数字编号)。PEP 是 Python Enhancement Proposals 的缩写，翻译过来就是 Python 增强建议书的意思。它是用来规范与定义 Python 的各种加强与延伸功能的技术规格，好让 Python 开发社区能有共同遵循的依据。

每个 PEP 都有一个唯一的编号，这个编号一旦给定了就不会再改变。例如，PEP 3000 就是用来定义 Python3 的相关技术规格；而 PEP 333 则是 Python 的 Web 应用程序界面 WSGI (Web Server Gateway Interface 1.0)的规范。关于 PEP 本身的相关规范是定义在 PEP 1，而 PEP 8 则定义了 Python 代码的风格指南。有关 PEP 的列表大家可以参考 PEP 0：https://www.python.org/dev/peps/。

举个例子，说说作者平时遇到问题是怎么自救的。前边不是举了一个计时器的例子嘛，那是自己写的一个计时器。其实在实际应用中，不建议大家自己动手写计时器，因为有很多未知的因素会影响到你的数据。所以建议用现成的模块——timeit 来对你的代码进行计时。

那现在假设我不知道 timeit 模块的用法，应该如何下手？

首先应该先查找帮助文档，可以使用文档的搜索或者索引功能——一般情况下输入关键词之后，文档第一个显示出来的内容就是你需要的，如图 13-2 所示。




图 13-2　如何在帮助文档中找到自己需要的内容

首先出现的是关于这个模块的介绍，如图 13-3 所示。

Python » 3.4.3 Documentation » The Python Standard Library previous | next | modules | index
» 27. Debugging and Profiling »

27.5. timeit — Measure execution time of small code snippets

**Source code:** Lib/timeit.py

This module provides a simple way to time small bits of Python code. It has both a *Command-Line Interface* as well as a *callable* one. It avoids a number of common traps for measuring execution times. See also Tim Peters' introduction to the "Algorithms" chapter in the *Python Cookbook*, published by O'Reilly.

图 13-3　timeit 模块

帮大家大概翻译下：

**timeit — Measure execution time of small code snippets**

**timeit 模块详解——准确测量小段代码的执行时间**

Source code：Lib/timeit. py（该模块所在的位置）

---

timeit 模块提供了测量 Python 小段代码执行时间的方法。它既可以在命令行界面直接使用，

也可以通过导入模块进行调用。该模块灵活地避开了测量执行时间时容易出现的错误。

接下来就是简单的使用方法介绍，如图 13-4 所示。

## 27.5.1. Basic Examples

The following example shows how the *Command-Line Interface* can be used to compare three different expressions:

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 23.2 usec per loop
```

This can be achieved from the *Python Interface* with:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

Note however that `timeit` will automatically determine the number of repetitions only when the command-line interface is used. In the *Examples* section you can find more advanced examples.

图 13-4　timeit 模块简单的使用方法介绍

接着是指出这个模块里边包含哪些类、函数、变量及其功能和用法。最后就是实际应用的例子。基本上所有的模块文档都会遵循这么一个顺序。如果你认为要快速学习一个模块都得读这么长的文档的话，那你还是“too young，too simple”了。

快速掌握一个模块的用法，可以利用 IDLE。先导入模块：

```
>>> import timeit
```

可以调用__doc__属性，查看这个模块的简介，可以用 print 把它带格式的打印出来：

```
>>> print(timeit.__doc__)
Tool for measuring execution time of small code snippets.

This module avoids a number of common traps for measuring execution
times. See also Tim Peters' introduction to the Algorithms chapter in
the Python Cookbook, published by O'Reilly.

Library usage: see the Timer class.

Command line usage:
    python timeit.py [-n N] [-r N] [-s S] [-t] [-c] [-p] [-h] [--] [statement]

Options:
```

```
  -n/--number N: how many times to execute 'statement' (default: see below)
  -r/--repeat N: how many times to repeat the timer (default 3)
  -s/--setup S: statement to be executed once initially (default 'pass')
  -p/--process: use time.process_time() (default is time.perf_counter())
  -t/--time: use time.time() (deprecated)
  -c/--clock: use time.clock() (deprecated)
  -v/--verbose: print raw timing results; repeat for more digits precision
  -h/--help: print this usage message and exit
  --: separate options from statement, use when statement starts with -
  statement: statement to be timed (default 'pass')

A multi-line statement may be given by specifying each line as a
separate argument; indented lines are possible by enclosing an
argument in quotes and using leading spaces. Multiple -s options are
treated similarly.

If -n is not given, a suitable number of loops is calculated by trying
successive powers of 10 until the total time is at least 0.2 seconds.

Note: there is a certain baseline overhead associated with executing a
pass statement. It differs between versions. The code here doesn't try
to hide it, but you should be aware of it. The baseline overhead can be
measured by invoking the program without arguments.

Classes:

    Timer

Functions:

    timeit(string, string) -> float
    repeat(string, string) -> list
    default_timer() -> float
```

使用dir()函数可以查询到该模块定义了哪些变量、函数和类：

```
>>> dir(timeit)
['Timer', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', '_template_func', 'default_number', 'default_repeat', 'default_timer', '
dummy_src_name', 'gc', 'itertools', 'main', 'reindent', 'repeat', 'sys', 'template', 'time', 'timeit']
```

但并不是所有这些名字对我们都有用，所以要过滤掉一些不需要的东西。你可能留意到这里有个__all__属性，事实是它就是帮助我们完成这么一个过滤的操作：

```
>>> timeit.__all__
['Timer', 'timeit', 'repeat', 'default_timer']
```

timeit模块其实只有一个类和三个函数供我们外部调用而已，所以用__all__属性就可以直接获得可供调用接口的信息。

这里有两点需要注意：第一，不是所有的模块都有__all__属性；第二，如果一个模块设置了__all__属性，那么使用“from timeit import *”这样的形式导入命名空间，就只有__all__属性这个列表里边的名字才会被导入，其他的名字不受影响：

```
>>> Timer
<class 'timeit.Timer'>
>>> gc
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    gc
NameError: name 'gc' is not defined
```

但如果没有设置__all__属性的话，用“from 模块名 import *”就会把所有不以下划线开头的名字都导入到当前的命名空间。所以，建议在编写模块的时候，将对外提供的接口函数和类都设置到__all__属性这个列表里。

另外还有一个叫作__file__的属性，这个属性指明了该模块的源代码位置：

```
>>> import timeit
>>> timeit.__file__
'C:\\Python34\\lib\\timeit.py'
```

最后，还有一道杀手锏，也是我们常用的——使用 help()函数：

```
>>> help(timeit)
# 太长…省略…
```

关于 timeit 模块，由于这个模块实在太有用了（经常用来实现代码计时），所以作者把对应的文档做了下翻译，大家可以收藏一下，今后你肯定会用上的（http://bbs.fishc.com/thread-55593-1-1.html）。

# 第14章 论一只爬虫的自我修养

## 14.1 入门



本章教大家写一只属于你自己的网络爬虫。那什么是网络爬虫呢？网络爬虫，又称为网页蜘蛛（WebSpider），非常形象的一个名字。如果你把整个互联网想象成类似于蜘蛛网一样的构造，那么这只爬虫，就是要在上边爬来爬去，以便捕获我们需要的资源。

我们之所以能够通过百度或谷歌这样的搜索引擎检索到你的网页，靠的就是他们大量的爬虫每天在互联网上爬来爬去，对网页中的每个关键词进行索引，建立索引数据库。在经过复杂的算法进行排序后，这些结果将按照与搜索关键词的相关度高低，依次排列。

当然，编写一个搜索引擎，是一件非常艰苦的事情……但千里之行，始于足下！先从编写一个小爬虫代码开始，然后不断地来改进它。

使用 Python 编写爬虫代码，要解决的第一个问题是：Python 如何访问互联网？

好现实的一个问题……

好在 Python 为此准备好了"电池"：urllib 模块。

事实上这个 urllib 是 URL 和 lib 两个单词共同构成的：URL，大家都知道，就是平时说的网页的地址；lib 是 library（库）的缩写。像鱼 C 工作室的首页，URL 的地址就是 http://www.fishc.com。

URL 的一般格式为（带方括号[]的为可选项）：protocol://hostname[port]/path/[;parameters][?query]#fragment。

URL 由三部分组成：

(1) 协议，常见的有 http、https、ftp、file（访问本地文件夹）、ed2k（电驴的专用链接）等等。

(2) 存放资源的服务器的域名系统（DNS）主机名或 IP 地址（有时候要包含端口号，各种传输协议都有默认的端口号，如 http 的默认端口为 80）。

(3) 主机资源的具体地址，如目录和文件名等。

第 1 部分和第 2 部分用"://"符号隔开，

第 2 部分和第 3 部分用"/"符号隔开。

第 1 部分和第 2 部分是不可缺少的，第 3 部分有时可以省略。

说完 URL，可以来谈这个 urllib 模块了。Python3 其实对这个模块做了挺大的改动，以前有一个 urllib 模块还有一个 urllib2 模块（对 urllib 的补充），乱得很……Python3 干脆将它们合并在了一起，统一叫 urllib。这其实也不是一个模块，它是一个包（package）。

打开参考文档看一下，如图 14-1 所示。

# 21.5. urllib — URL handling modules

urllib is a package that collects several modules for working with URLs:

- urllib.request for opening and reading URLs
- urllib.error containing the exceptions raised by urllib.request
- urllib.parse for parsing URLs
- urllib.robotparser for parsing robots.txt files

图 14-1　urllib 模块

其实 urllib 是一个包，里边总共有四个模块。第一个模块是最复杂的也是最重要的，因为它包含了对服务器请求的发出、跳转、代理和安全等各个方面。

先来体验一下吧，通过 urllib.request.urlopen()函数就可以访问网页了：

```
>>> import urllib.request
>>> response = urllib.request.urlopen("http://www.fishc.com")
>>> html = response.read()
>>> print(html)
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN"\r\n\t"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">\r\n\r\n<!-- \r\n(c) 2011
\xc4\xbdubom\xc3\xadr Krupa, CC BY-ND 3.0\r\n -->\t\r\n\r\n<html
xmlns="http://www.w3.org/1999/xhtml">\r\n\t<head>\r\n\t\t<meta
...
```

细心的读者可能会发现，这跟在浏览器上使用“审查元素”功能看到的内容不太一样？如图 14-2 所示。

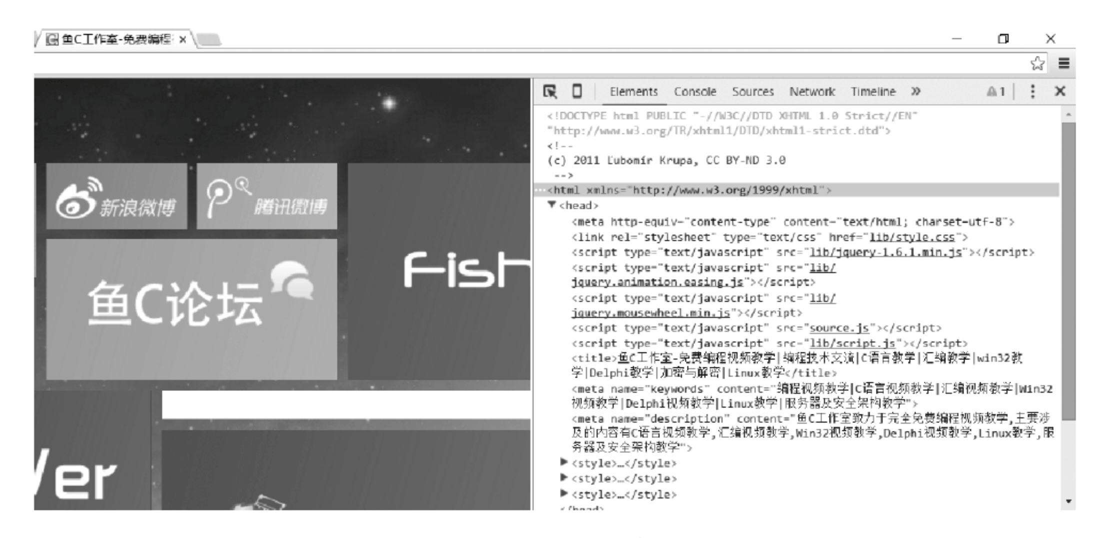



图 14-2　审查元素

其实 Python 爬取的是内容是以 utf-8 编码的 bytes 对象(注意：上边打印的字符串前边有个 b，表示这是一个 bytes 对象，可以理解为字符串的每个字符用于存放一个字节的二进制

数据)，要还原为带中文的 html 代码，需要对其进行解码，将它变成 Unicode 编码：

```
>>> html = html.decode("utf-8")
>>> print(html)
…
        <title>鱼C工作室 - 免费编程视频教学|编程技术交流|C语言教学|汇编教学|win32教学|
Delphi 教学|加密与解密|Linux 教学</title>
…
```

### 什么是编码

事实上计算机只认识 0 和 1，然而却可以通过计算机来显示文本，这就是靠编码实现的。编码其实就是约定的一个协议，比如 ASCII 编码约定了大写字母 A 对应十进制数 65，那么在读取一个字符串的时候，看到 65，计算机就知道这是大写字母 A 的意思。

由于计算机是美国人发明的，所以这个 ASCII 编码设计时只采用 1 个字节存储，包含了大小写英文字母、数字和一些符号。但是计算机在全世界普及之后，ASCII 编码就成了一个瓶颈，因为 1 个字节是完全不足以表示各国语言的。

大家都知道英文只用 26 个字母就可以组成不同的单词，而汉字光常用字就有好几千个，至少需要 2 个字节才足以存放，所以后来中国制定了 GB 2312 编码，用于对汉字进行编码。

然后日本为自己的文字制定了 Shift_JIS 编码，韩国为自己的文字制定了 Euc-kr 编码，一时之间，各国都制定了自己的标准。不难想象，不同的标准放在一起，就难免出现冲突。这也正是为什么最初的在计算机上总是容易看到乱码的现象。

为了解决这个问题，Unicode 编码应运而生。Unicode 组织的想法最初也很简单：创建一个足够大的编码，将所有国家的编码都加进来，进行统一标准。

没错，这样问题就解决了。但新的问题也出现了：如果你写的文本只包含英文和数字，那么用 Unicode 编码就显得特别浪费存储空间(用 ASCII 编码只占用一半的存储空间)。所以本着能省一点是一点的精神，Unicode 还创造出了多种实现方式。

比如常用的 UTF-8 编码就是 Unicode 的一种实现方式，它是可变长编码。简单地说，就是当文本是 ASCII 编码的字符时，它用 1 个字节存放；而当文本是其他 Unicode 字符的情况，它将按一定算法转换，每个字符使用 1～3 个字节存放。这样便实现了有效节省空间的目的(注：有关编码的详细介绍，可以参考 http://bbs.fishc.com/thread-46797-1-1.html)。

## 14.2　实战



### 14.2.1　下载一只猫

第一个例子是"下载一只猫"，这是 codecademy 上边的一个例子。我们说林子大了，什么鸟都有。互联网这么大，想当然也有各种不同特色的网站。第一个例子需要访问 http://placekitten.com 这个网站，这是一个为"猫奴"量身定制的站点。

在网址后边直接附上宽度和高度，就可以得到一张对应的猫的图片，例如访问的地址是 http://placekitten.com/g/200/300，那么将得到一张宽度为 200 像素、高度为 300 像素的图片，如图 14-3 所示。




图 14-3　获得喵星人的照片

获取的图片都是.jpg 格式的,你可以使用右键快捷菜单中的“图片另存为”命令将其直接保存到本地。

现在用 Python 来实现刚才的操作:

```
# p14_1.py
import urllib.request

response = urllib.request.urlopen("http://placekitten.com/g/200/300")
cat_img = response.read()
with open('cat_200_300.jpg', 'wb') as f:
    f.write(cat_img)
```

快看,代码所在的文件夹中是不是出现了 cat_200_300.jpg 这张图片?

不错,既然程序可以顺利执行,那接下来快速地解读一下代码,避免大家有些地方理解不到位:首先,urlopen 的 url 参数既可以是一个字符串也可以是 Request 对象,如果你传入一个字符串,那么 Python 是会默认先帮你把目标字符串转换成 Request 对象,然后再传给 urlopen 函数。

因此,代码也可以这么写:

```
…
req = urllib.requset.Requset("http://placekitten.com/g/200/300")
response = urllib.request.urlopen(req)
…
```

然后,urlopen 实际上返回的是一个类文件对象,因此你可以用 read()方法来读取内容。除此之外,文档还告诉你以下三个函数可能以后会用到:

- geturl()——返回请求的 url。
- info()——返回一个 httplib.HTTPMessage 对象,包含远程服务器返回的头信息。
- getcode()——返回 HTTP 状态码。



### 14.2.2　翻译文本

第二个例子要求利用有道词典来翻译文本。

首先来到官网(http://www.youdao.com),单击“有道翻译”图标(http://fanyi.youdao.com),出现如图 14-4 所示的界面。




图 14-4　有道翻译

首先使用浏览器的“审查元素”功能(现在基本上所有的浏览器都自带有这么一个调试插件),这里使用的是谷歌浏览器,其他浏览器的用法也差不多。在浏览器中右击,选择“审查元素”命令,切换到 Network 窗口,如图 14-5 所示。




图 14-5　“审查元素”的使用(一)

这时候单击页面中的“自动翻译”按钮,就会发现拦截到许多文件,这些文件就是浏览器和客户端的通信内容,如图 14-6 所示。

在客户端和服务器之间进行请求-响应时,两种最常被用到的方法是 GET 和 POST。通常,GET 是从指定的服务器请求数据。而 POST 是向指定的服务器提交要被处理的数据(当然,这不是绝对的,因为在现实情况中,GET 也用来提交数据给服务器)。

那刚才的动作是提交数据,对吧？所以一眼看到 POST,赶紧打开来看看,如图 14-7 所示。

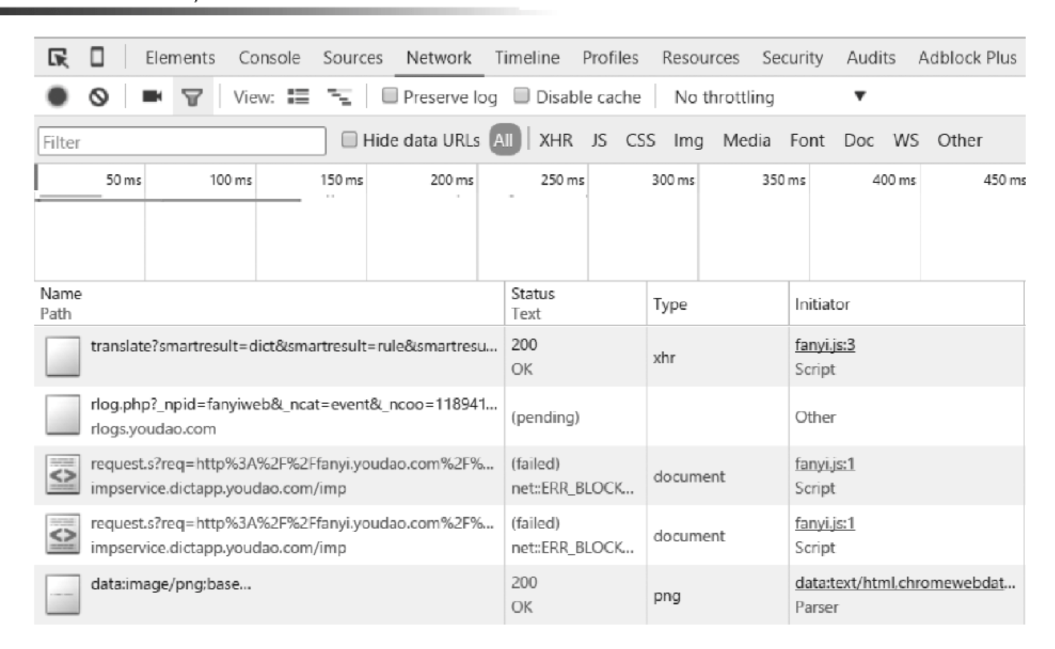



图 14-6 “审查元素”的使用(二)

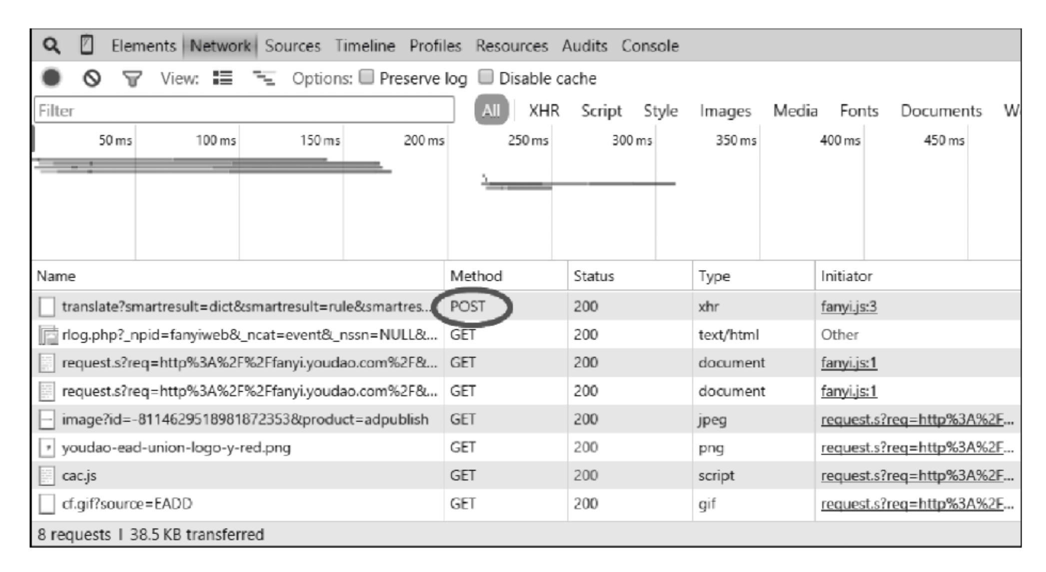



图 14-7 “审查元素”的使用(三)

单击“translate? smartresult = dict&smartresult = rule&smartres…”选项，如图 14-8 所示。

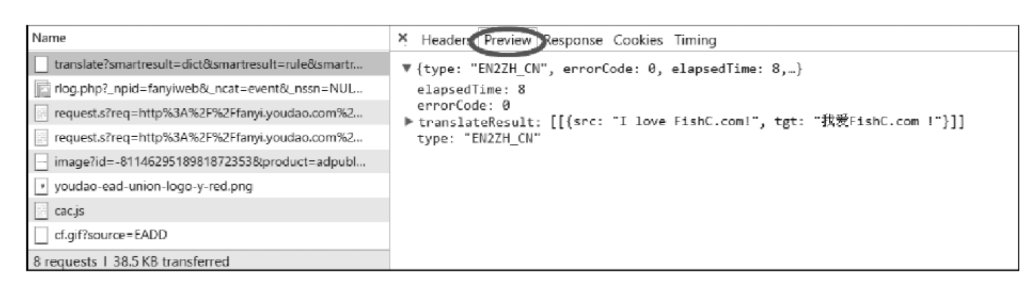



图 14-8 “审查元素”的使用(四)

运气很好，一下子就找到了，这正是我们需要的内容！选择 Headers 选项卡，如图 14-9 所示。



图 14-9 “审查元素”的使用（五）

HTTP 是基于请求-响应的模式的，客户端发出的请求叫 Request，服务端的响应叫 Response。下面针对一些关键名词，给大家做下注释：

```
Remote Address: 61.135.218.44:80
# 服务器 ip 地址和端口
Request URL: http://fanyi.youdao.com/translate?smartresult = dict&smartresult = rule&ssmartresul =
ugc&sessionFrom = http://www.youdao.com/
# 请求的链接地址
Request Method: POST
# 请求的方法,这里是 POST
Status Code: 200 OK
# 状态码,200 表示正常响应
```

Request Headers 是客户端发送请求的 Headers，这个常常被服务端用来判断是否来自“非人类”的访问。什么意思呢？例如写个 Python 代码，然后用这个代码批量访问网站的数据，这样服务器压力就会增大，所以一般服务器不欢迎“非人类”的访问。

一般是通过这个 User-Agent 来识别，普通浏览器会通过该内容向访问网站提供你所使用的浏览器类型、操作系统、浏览器内核等信息的标识：

```
User - Agent:
Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.65
Safari/537.36
```

而使用 Python 访问的话，User-Agent 会被定义为 Python-urllib/3.4。

所以检查这个就可以查到请求是来自正常的浏览器单击还是“非人类”的访问。当然，如果服务器以为这样就能阻挡我们前进的脚步，那它就太天真了。这个 User-Agent 其实是可以自定义的，后边再给大家介绍。

除此之外不难发现，下边有一个 Form Data，这个就是 POST 提交的内容（大家看，里边不正有“I love FishC.com!”嘛）。

最后一个问题，那如何用 Python 提交 POST 表单呢？看文档，如图 14-10 所示。

这里说得很仔细了：urlopen 函数有一个 data 参数，如果给这个参数赋值，那么 HTTP 的请求就是使用 POST 方式；如果 data 的值是 NULL，也就是默认值，那么 HTTP 的请求就是

urllib.request.**urlopen**(*url*, *data=None*, [*timeout*, ]*, *cafile=None*, *capath=None*, *cadefault=False*, *context=None*)

Open the URL *url*, which can be either a string or a Request object.

*data* must be a bytes object specifying additional data to be sent to the server, or None if no such data is needed. *data* may also be an iterable object and in that case Content-Length value must be specified in the headers. Currently HTTP requests are the only ones that use *data*; the HTTP request will be a POST instead of a GET when the *data* parameter is provided.

*data* should be a buffer in the standard *application/x-www-form-urlencoded* format. The urllib.parse.urlencode() function takes a mapping or sequence of 2-tuples and returns a string in this format. It should be encoded to bytes before being used as the *data* parameter. The charset parameter in Content-Type header may be used to specify the encoding. If charset parameter is not sent with the Content-Type header, the server following the HTTP 1.1 recommendation may assume that the data is encoded in ISO-8859-1 encoding. It is advisable to use charset parameter with encoding used in Content-Type header with the Request.

图 14-10　urllib.request.urlopen()提交数据

使用 GET 方式。这里还告诉我们了，这个 data 参数的值必须符合这个 application/x-www-form-urlencoded 的格式。然后还不厌其烦地告诉我们要用 urllib.parse.urlencode()将字符串转换为这个格式。

有了这两点知识其实就够了，来尝试写代码：

```
# p14_2.py
import urllib.request
import urllib.parse

url = "http://fanyi.youdao.com/translate?smartresult=dict&smartresult=rule&smartresult=ugc&sessionFrom=http://www.youdao.com/"
data = {}
data['type'] = 'AUTO'
data['i'] = 'I love FishC.com!'
data['doctype'] = 'json'
data['xmlVersion'] = '1.6'
data['keyfrom'] = 'fanyi.web'
data['ue'] = 'UTF-8'
data['typoResult'] = 'true'
data = urllib.parse.urlencode(data).encode('utf-8')
response = urllib.request.urlopen(url, data)
html = response.read().decode('utf-8')
print(html)
```

程序执行结果如下：

```
>>>
{"type":"EN2ZH_CN","errorCode":0,"elapsedTime":2,"translateResult":[[{"src":"I love FishC.com!","tgt":"我爱 FishC.com !"}]]}
```

字符串在 Python 内部的表示是 Unicode 编码，因此，在做编码转换时，通常需要以 Unicode 作为中间编码，即先将返回的 bytes 对象的数据解码(decode)成 Unicode，再从 Unicode 编码(encode)成另一种编码。

有关编码的问题探讨，大家还可以参考以下这篇文章，相信会使你受益匪浅的：Python 编码问题的解决方案总结(http://bbs.fishc.com/thread-56452-1-1.html)。

数据是得到了，但是这样显示未免也太丑了吧！那怎么办呢？这是一个字符串，我们可以用查找字符串的方式把需要的内容给显示出来。但我们不会这么做，因为这也太被动了……

这其实是一个 JSON 格式的字符串(JSON 是一种轻量级的数据交换格式，说白了这里就是用字符串把 Python 的数据结构封装起来)，所以只需要解析这个 JSON 格式的字符串即可：

```
>>> import json
>>> json.loads(html)
{'translateResult': [[{'tgt': '我爱 FishC.com！', 'src': 'I love FishC.com!'}]], 'type': 'EN2ZH_CN',
'errorCode': 0, 'elapsedTime': 2}
```

可以看到它已经变成一个字典了，接下来的事儿就好办多了：

```
>>> target = json.loads(html)
>>> type(target)
<class 'dict'>
>>> target['translateResult'][0][0]['tgt']
'我爱 FishC.com！'
```

最后让我们把程序美化一下：

```
# p14_3.py
import urllib.request
import urllib.parse
import json

content = input("请输入需要翻译的内容：")
url = "http://fanyi.youdao.com/translate?smartresult=dict&smartresult=rule&smartresult=
ugc&sessionFrom=http://www.youdao.com/"
data = {}
data['type'] = 'AUTO'
data['i'] = content
data['doctype'] = 'json'
data['xmlVersion'] = '1.6'
data['keyfrom'] = 'fanyi.web'
data['ue'] = 'UTF-8'
data['typoResult'] = 'true'
data = urllib.parse.urlencode(data).encode('utf-8')
response = urllib.request.urlopen(url, data)
html = response.read().decode('utf-8')
target = json.loads(html)
print("翻译结果：%s" % (target['translateResult'][0][0]['tgt']))
```

程序执行结果如下：

```
>>>
请输入需要翻译的内容：小甲鱼
```

```
翻译结果: The little turtle
>>>
```

这里需要注意的是：这样的代码你还不能应用到现实的生产环境中，因为服务器一“看”User-Agent来源是“非人类”，它就把你屏蔽了。或者一看你这IP怎么访问这么频繁，也给你拉黑……

## 14.3 隐藏



有些网站不喜欢被程序访问，因此它们会检查链接的来源。如果访问来源不是正常的途径，就给你“掐掉”。所以为了让我们的爬虫更好地为我们服务，需要对代码进行一些改进——隐藏，让它看起来更像是普通人通过普通浏览器的正常点击。

### 14.3.1 修改 User-Agent

在文档中搜索User-Agent，可以看到urllib.request.Request部分有关于设置User-Agent的叙述，如图14-11所示。

*headers* should be a dictionary, and will be treated as if `add_header()` was called with each key and value as arguments. This is often used to "spoof" the `User-Agent` header, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`, while `urllib`'s default user agent string is `"Python-urllib/2.6"` (on Python 2.6).

图14-11 设置User-Agent

文档中说得很清楚了：Request有个headers参数，通过设置这个参数，你可以伪造成浏览器访问。设置这个headers参数有两种途径：实例化Request对象的时候将headers参数传进去和通过add_header()方法往Request对象添加headers。

第一种方法要求headers必须是一个字典的形式：

```
# p14_4.py
import urllib.request
import urllib.parse
import json

content = input("请输入需要翻译的内容：")
url = "http://fanyi.youdao.com/translate?smartresult = dict&smartresult = rule&smartresult = ugc&sessionFrom = http://www.youdao.com/"
head = {}
head['Referer'] = 'http://fanyi.youdao.com'
head['User - Agent'] = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36X - Requested - With:XMLHttpRequest'
data = {}
data['type'] = 'AUTO'
data['i'] = content
```

```
data['doctype'] = 'json'
data['xmlVersion'] = '1.6'
data['keyfrom'] = 'fanyi.web'
data['ue'] = 'UTF-8'
data['typoResult'] = 'true'
data = urllib.parse.urlencode(data).encode('utf-8')

req = urllib.request.Request(url, data, head)
response = urllib.request.urlopen(req)
html = response.read().decode('utf-8')
target = json.loads(html)
print("翻译结果: %s" % (target['translateResult'][0][0]['tgt']))
```

测试下是否成功修改：

```
>>>
请输入需要翻译的内容: 爱情
翻译结果: love
>>> req.headers
{'Referer': 'http://fanyi.youdao.com', 'User-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36X-Requested-With:XMLHttpRequest'}
>>>
```

还有另外一种方法，就是在 Request 对象生成之后，用 add_header()方法追加进去：

```
# p14-5.py
import urllib.request
import urllib.parse
import json

content = input("请输入需要翻译的内容: ")
url = "http://fanyi.youdao.com/translate?smartresult=dict&smartresult=rule&smartresult=ugc&sessionFrom=http://www.youdao.com/"
data = {}
data['type'] = 'AUTO'
data['i'] = content
data['doctype'] = 'json'
data['xmlVersion'] = '1.6'
data['keyfrom'] = 'fanyi.web'
data['ue'] = 'UTF-8'
data['typoResult'] = 'true'
data = urllib.parse.urlencode(data).encode('utf-8')
req = urllib.request.Request(url, data)
req.add_header('Referer', 'http://fanyi.youdao.com')
req.add_header('User-Agent', 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36X-Requested-With:XMLHttpRequest')
response = urllib.request.urlopen(req)
html = response.read().decode('utf-8')
target = json.loads(html)
print("翻译结果: %s" % (target['translateResult'][0][0]['tgt']))
```

测试下是否成功修改：

```
>>>
请输入需要翻译的内容: love
翻译结果: 爱
```

```
>>> req.headers
{'User-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/39.0.2171.95 Safari/537.36X-Requested-With: XMLHttpRequest', 'Referer':
'http://fanyi.youdao.com'}
>>>
```

通过修改 User-Agent 实现隐藏，可以算是最简单的方法了。不过如果这是一个用于抓取网页的爬虫（例如说批量下载某些图片……），那么一个 IP 地址在短时间内连续进行网页访问，很明显是不符合普通人类的行为标准的，同时也会对服务器造成不小的压力。因此服务器只需要记录每个 IP 的访问频率，在单位时间之内，如果访问频率超过一个阈值，便认为该 IP 地址很可能是爬虫，于是可以返回一个验证码页面，要求用户填写验证码。如果是爬虫的话，当然不可能填写验证码，便可拒绝掉。

那怎么办呢？那就我们目前的知识水平，有两种策略可供选择：第一种就是延迟提交的时间，还有一种策略就是使用代理。

## 14.3.2 延迟提交数据

延迟提交的时间，这个容易，用 time 模块的即可：

```
# p14_6.py
import urllib.request
import urllib.parse
import json
import time

url = "http://fanyi.youdao.com/translate?smartresult=dict&smartresult=rule&smartresult=
ugc&sessionFrom=http://www.youdao.com/"
while True:
    content = input('请输入待翻译内容(输入"q!"退出程序): ')
    if content == 'q!':
        break
    data = {}
    data['type'] = 'AUTO'
    data['i'] = content
    data['doctype'] = 'json'
    data['xmlVersion'] = '1.6'
    data['keyfrom'] = 'fanyi.web'
    data['ue'] = 'UTF-8'
    data['typoResult'] = 'true'
    data = urllib.parse.urlencode(data).encode('utf-8')
    req = urllib.request.Request(url, data)
    req.add_header('Referer', 'http://fanyi.youdao.com')
    req.add_header('User-Agent', 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/
537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36X-Requested-With:XMLHttpRequest')
    response = urllib.request.urlopen(req)
    html = response.read().decode('utf-8')
    target = json.loads(html)
    print("翻译结果: %s" % (target['translateResult'][0][0]['tgt']))
    time.sleep(5)
```

如果这样做，会使得程序的工作效率降低。

### 14.3.3　使用代理

第二个方案是使用代理。代理是什么？代理就是“嘿，兄弟，哥们访问这网址有点困难，帮忙解决一下呗。”然后就把需要访问的网址告诉代理，代理替你访问，然后把看到的内容都转发给你，这就是代理的工作。因此服务器看到的是代理的 IP 地址，而不是你的 IP 地址。

使用代理的步骤如下：

（1）proxy_support = urllib.request.ProxyHandler({})

＃参数是一个字典，字典的键是代理的类型，例如 http、ftp 或 https，字典的值就是代理的 IP 地址和对应的端口号。

（2）opener = urllib.request.build_opener(proxy_support)

＃这里大家先要知道什么是 opener？

＃ opener 可以看作是一个私人定制，当使用 urlopen()函数打开一个网页的时候，你就是使用默认的 opener 在工作。

＃而这个 opener 是可以定制的，例如，给它定制特殊的 headers，或者给它定制指定的代理 IP

＃所以这里使用 build_opener()函数创建了一个属于我们自己私人定制的 opener

（3）urllib.request.install_opener(opener)

＃这里是将定制好的 opener 安装到系统中，这是一劳永逸的做法。

＃因为在此之后，你只要使用普通的 urlopen()函数，就是以定制好的 opener 进行工作的。

＃如果你不想替换掉默认的 opener，你也可以在每次特殊需要的时候，用 opener.open()的方法来打开网页。

举个例子：搜索“代理 IP”字样获得一个免费的代理 IP 地址（这里找到的代理 IP 是：211.138.121.38:80），通过访问 http://www.whatismyip.com.tw 可以查看当前 IP。

```
# p14_7.py
import urllib.request

url = 'http://www.whatismyip.com.tw/'
proxy_support = urllib.request.ProxyHandler({'http':'211.138.121.38:80'})
# 接着创建一个包含代理 IP 的 opener
opener = urllib.request.build_opener(proxy_support)
# 安装进默认环境
urllib.request.install_opener(opener)
# 试试看 IP 地址改了没
response = urllib.request.urlopen(url)
html = response.read().decode('utf-8')
print(html)
```

让程序跑起来吧：

```
>>>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
```

```
    <meta name="description" content="我的 IP 查询"/>
    <meta name="keywords" content="查 IP,IP 查询,查我的 IP,我的 IP 地址,我的 IP 位置,侦测我
的 IP,查询我的 IP,查看我的 IP,显示我的 IP,what is my IP,whatismyip,my IP address,my IP proxy"/>
    <title>我的 IP 位址查询</title>
  </head>
  <body>
<h1>IP 位址</h1><h2>211.138.121.38</h2>
…
```

还可以设置一个 iplist，多填写几个 IP 进去，然后每次随机使用一个 IP 来访问：

```
# p14_8.py
import urllib.request
import random

url = 'http://www.whatismyip.com.tw/'
print("添加代理 IP 地址(IP:端口号),多个 IP 地址间用英文的分号隔开!")
iplist = input("请开始输入: ").split(sep=";")
while True:
    ip = random.choice(iplist)
    proxy_support = urllib.request.ProxyHandler({'http':ip})
    opener = urllib.request.build_opener(proxy_support)
    opener.addheaders = [('User-Agent', 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36')]
    urllib.request.install_opener(opener)
    try:
        print("正在尝试使用 %s 访问..." % ip)
        response = urllib.request.urlopen(url)
    except urllib.error.URLError:
        print("访问出错!")
    else:
        print("访问成功!")
    if input("请问是否继续?(Y/N)") == 'N':
        break
```

让程序跑起来吧：

```
>>>
添加代理 IP 地址(IP:端口号),多个 IP 地址间用英文的分号隔开!
请开始输入: 124.254.57.150:8118;61.184.192.42:80;88.88.88.88:88
正在尝试使用 61.184.192.42:80 访问...
访问成功!
请问是否继续?(Y/N)Y
正在尝试使用 61.184.192.42:80 访问...
访问成功!
请问是否继续?(Y/N)Y
正在尝试使用 88.88.88.88:88 访问...
访问出错!
请问是否继续?(Y/N)Y
正在尝试使用 124.254.57.150:8118 访问...
访问成功!
请问是否继续?(Y/N)N
>>>
```



# 14.4 Beautiful Soup



这一讲给大家介绍一个压箱底的模块——Beautiful Soup 4，翻译过来名字有点诡异：漂亮的汤？美味的鸡汤？好吧，只要你写出一个普罗大众都喜欢的模块，你管它叫"Beautiful Boy"大家也是能接受的……

Beautiful Soup 是一个可以从 HTML 或 XML 文件中提取数据的 Python 库。它能够通过你喜欢的转换器实现常规的文档导航、查找、修改文档的操作。Beautiful Soup 会帮你节省数小时甚至数天的工作时间。

安装 Beautiful Soup 非常容易，打开命令行窗口(CMD)，输入 py-3-m pip install BeautifulSoup4 命令，如图 14-12 所示。

```
C:\Users\佳宇>py -3 -m pip install BeautifulSoup4
Collecting BeautifulSoup4
  Downloading beautifulsoup4-4.4.1-py3-none-any.whl (81kB)
    100% |████████████████████████████████| 81kB 1.2MB/s
Installing collected packages: BeautifulSoup4
Successfully installed BeautifulSoup4-4.4.1
```

图 14-12　使用 pip 安装 Beautiful Soup 4

安装好了，该如何使用呢？基本的用法可以参考官方的快速入门文档(网址 https://www.crummy.com/software/BeautifulSoup/bs4/doc.zh/index.html)。下面通过几个案例给大家演示如何将其应用到爬虫中。

案例一：编写一个爬虫，爬取百度百科"网络爬虫"的词条(网址 http://baike.baidu.com/view/284853.htm)，并将所有包含"view"关键字的链接按格式打印出来，如图 14-13 所示。

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
锁定 -> http://baike.baidu.com/view/10812319.htm
网络爬虫 -> http://baike.baidu.com/view/284853.htm
FOAF -> http://baike.baidu.com/view/271451.htm
万维网 -> http://baike.baidu.com/view/7833.htm
蠕虫 -> http://baike.baidu.com/view/2596.htm
万维网 -> http://baike.baidu.com/view/7833.htm
网络 -> http://baike.baidu.com/view/3487.htm
```

图 14-13　爬取百度百科"网络爬虫"的词条

因为 Beautiful Soup 是用于从 HTML 或 XML 文件中提取数据，所以需要先使用 urllib.request 模块从指定网址上先读取 HTML 文件：

```
>>> import urllib.request
>>> from bs4 import BeautifulSoup
>>> url = "http://baike.baidu.com/view/284853.htm"
>>> response = urllib.request.urlopen(url)
>>> html = response.read()
>>> soup = BeautifulSoup(html, "html.parser")
```

BeautifulSoup(html，“html. parser”)需要两个参数：第一参数是需要提取数据的HTML或XML文件，第二个参数是指定解析器。然后使用find_all(href=re. compile(“view”))方法可以读取所有包含“view”关键字的链接(这里使用到正则表达式的知识，在14.5节中会详细讲解)，使用for语句迭代读取：

```
>>> import re
>>> for each in soup.find_all(href = re.compile("view")):
     print(each.text, " ->",''.join(["http://baike.baidu.com", \
each["href"]]))

锁定 -> http://baike.baidu.com/view/10812319.htm
网络爬虫 -> http://baike.baidu.com/view/284853.htm
FOAF -> http://baike.baidu.com/view/271451.htm
万维网 -> http://baike.baidu.com/view/7833.htm
蠕虫 -> http://baike.baidu.com/view/2596.htm
…
```

将代码整合一下，得到以下清单：

```
# p14_91.py
import urllib.request
import re
from bs4 import BeautifulSoup

def main():
    url = "http://baike.baidu.com/view/284853.htm"
    response = urllib.request.urlopen(url)
    html = response.read()
    soup = BeautifulSoup(html, "html.parser")

    for each in soup.find_all(href = re.compile("view")):
        print(each.text, " ->", ''.join(["http://baike.baidu.com", \
each["href"]]))

if __name__ == "__main__":
    main()
```

直接打印词条名和链接不算什么真本事，第二个案例要求爬虫允许用户输入搜索的关键词，可以进入每一个词条，然后检测该词条是否具有副标题(比如搜索“猪八戒”，副标题就是“(中国神话小说《西游记》的角色)”)，如果有，请将副标题一并打印出来，如图14-14所示。

图14-14　百度百科“猪八戒”的词条

要求程序实现如图 14-15 所示。

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
请输入关键词：猪八戒
多义词目录 -> http://baike.baidu.com/view/10812277.htm
义项目录 -> http://baike.baidu.com/view/340519.htm
共5个义项 -> http://baike.baidu.com/view/17156.htm?force=1
日本动漫《最游记》人物（日本动漫《最游记》人物） -> http://baike.baidu.com/subview/17156/503
9854.htm#viewPageContent
歌手张羽伟专辑（歌手张羽伟专辑） -> http://baike.baidu.com/subview/17156/5039855.htm#viewPa
geContent
《乱斗西游》人物设定（《乱斗西游》人物设定） -> http://baike.baidu.com/item/%E7%8C%AA%E5%85%
AB%E6%88%92/17330461#viewPageContent
动画电影《西游记之大圣归来》中的角色（动画电影《西游记之大圣归来》中的角色） -> http://baike.ba
idu.com/item/%E7%8C%AA%E5%85%AB%E6%88%92/18290279#viewPageContent
锁定 -> http://baike.baidu.com/view/10812319.htm
吴承恩（西汉将领） -> http://baike.baidu.com/subview/5787/19006941.htm
西游记（中国古典长篇小说） -> http://baike.baidu.com/view/2583.htm
猪刚鬣（中国神话小说《西游记》的角色） -> http://baike.baidu.com/view/541935.htm
唐僧（《西游记》中的人物） -> http://baike.baidu.com/view/17148.htm
```

如果存在副标题，
请将它们打印出来

图 14-15　按要求打印词条

代码清单如下：

```
# p14_92.py
import urllib.request
import urllib.parse
import re
from bs4 import BeautifulSoup

def main():
    keyword = input("请输入关键词:")
    keyword = urllib.parse.urlencode({"word":keyword})
    response = \
urllib.request.urlopen("http://baike.baidu.com/search/word?%s" % \
keyword)
    html = response.read()
    soup = BeautifulSoup(html, "html.parser")

    for each in soup.find_all(href=re.compile("view")):
        content = ''.join([each.text])
        url2 = ''.join(["http://baike.baidu.com", each["href"]])
        response2 = urllib.request.urlopen(url2)
        html2 = response2.read()
        soup2 = BeautifulSoup(html2, "html.parser")
        if soup2.h2:
            content = ''.join([content, soup2.h2.text])
        content = ''.join([content, " -> ", url2])
        print(content)

if __name__ == "__main__":
    main()
```

## 14.5 正则表达式



那么对于字符串查找，我想你应该已经是深恶痛绝了……

不难发现，下载一个网页是容易的，但是要在网页中找到你需要的内容，那是困难的！你发现字符串查找并不是那么简单，并不是直接使用 find()方法就能找到字符串位置就可以了。

通过前面的学习，你肯定尝试过写一个脚本来自动获取最新的代理 IP 地址，但是你肯定也遇到了字符串查找上的困难。好，那现在来重现一下大家可能遇到的困难：

目标 URL：http://cn-proxy.com/。

首先踩点，如图 14-16 所示。




图 14-16　踩点

读者朋友可能发现这回很难定位到 IP 以及对应端口的位置了。因为对于这个网页似乎找不到“唯一”的特征。看来看去只有 class="sortable"似乎是两个 ip 表格唯一的“特性”，因此你写了段代码，先搜索 class="sortable"关键字，然后再往下找到第六个＜td＞标签，总算是成功地定位到第一个 IP 地址……

这样写代码不仅费劲，而且难看，还不具备通用性。更惨的是，万一站长哪天心血来潮改了一下网页代码，你更是心塞啊！这时候你就会琢磨，可不可以按照我需要的内容特征进行查找呢？也就是说，比如我要找的是 IP 地址，那 IP 地址的特征就是有四段数字组成，每段数字的范围是 0～255，分别由三个点号(.)隔开。

没错，我们现在能够想到的，计算机的老前辈们也已经想到了，并且帮我们设计出了非常优秀的解决方案。就是我们今天要讲的是正则表达式。下面小甲鱼就教大家使用正则表达式来匹配 IP 地址。

关于正则表达式，有一个非常经典的美式笑话——有些人面临一个问题的时候会想：“我知道，可以使用正则表达式来解决这个问题。”于是，现在他就有两个问题了。有些读者朋友可能没懂，意思就是使用正则表达式，本身就是一个难题。

没错，正则表达式的确很难学，但却非常有用。这么说吧，在编写处理字符串的程序或网页时，经常会有查找符合某些复杂规则的字符串的需要。用 Python 自带的字符串方法，一定会让你恼羞成怒。这时候，如果你懂得正则表达式，你会发现这真是灵丹妙药，因为正则表达式就是用于描述这些复杂规则的工具。

### 14.5.1　re 模块

不同的语言均有使用正则表达式的方法，但各不相同。Python 是通过 re 模块来实现的。接下来，我们边写例子边给大家讲解，这样比较容易理解：

```
>>> import re
>>> re.search(r'FishC', 'I love FishC.com!')
<_sre.SRE_Match object; span=(7, 12), match='FishC'>
```

search()方法用于在字符串中搜索正则表达式模式第一次出现的位置，这里找到了，匹配的位置是(7，12)。

这里需要注意两点：

(1) 第一个参数是正则表达式模式，也就是你要描述的搜索规则，需要使用原始字符串来写，因为这样可以避免很多不必要的麻烦。

(2) 找到后返回的范围是以下标 0 开始的，这跟字符串一样。如果找不到，它就返回 None。

### 14.5.2　通配符

有些读者朋友可能会产生质疑了："你说了那么多，我用 find()方法不一样可以实现吗？"

```
>>> "I love FishC.com!".find('FishC')
7
```

好，那来一个 find()方法没法实现的……

大家都知道通配符，就是 * 和?，用它来表示任何字符。例如想找到所有 Word 类型的文件时，我们就输入 *.docx，对不对？

正则表达式也有所谓的通配符，在这里是用一个点号(.)来表示，它可以匹配除了换行符之外的任何字符：

```
>>> re.search(r'.', 'I love FishC.com!')
<_sre.SRE_Match object; span=(0, 1), match='I'>
>>> re.search(r'Fish.', 'I love FishC.com!')
<_sre.SRE_Match object; span=(7, 12), match='FishC'>
```

### 14.5.3　反斜杠

喜欢思考的读者朋友现在可能会有疑问了："既然点号(.)可以匹配任何字符，那如果现在只想单单匹配点号(.)这个字符本身，要怎么办呢？"

正如 Python 的字符串规则，想要消除一个字符的特殊功能，就在它前边加上反斜杠，这里也一样：

```
>>> re.search(r'.', 'I love FishC.com!')
<_sre.SRE_Match object; span=(0, 1), match='I'>
>>> re.search(r'\.', 'I love FishC.com!')
<_sre.SRE_Match object; span=(12, 13), match='.'>
```

在正则表达式中，反斜杠可以剥夺元字符(注：元字符就是拥有特殊能力的符号，像刚才的点号(.)就是一个元字符)的特殊能力。同时，反斜杠还可以使得普通字符拥有特殊能力。

举个例子，例如想匹配数字，那么可以使用反斜杠加上小写字母 d(\d)：

```
>>> re.search(r'\d', 'I love 123 FishC.com!')
<_sre.SRE_Match object; span=(7, 8), match='1'>
```

有了以上两点知识，要匹配一个 IP 地址大概就可以这么写：

```
>>> re.search(r'\d\d\d\.\d\d\d\.\d\d\d\.\d\d\d', 'other192.168.111.253other')
<_sre.SRE_Match object; span=(5, 20), match='192.168.111.253'>
```

当然这么写是有问题的：首先，\d 表示匹配 0～9 所有的数字，而 IP 地址约定的范围是 0～255，\d\d\d 表示的范围则是 000～999；其次，你这里要求 IP 地址的每个组成部分都需要三个数字，而现实中经常是像 192.168.1.1 这样；最后，这样的正则表达式未免也太丑了吧？！

既然有问题，那就有解决的方案，下边逐个来解决！

### 14.5.4 字符类

为了表示一个字符的范围，可以创建一个字符类。使用中括号将任何内容包起来就是一个字符类，它的含义是你只要匹配这个字符类中的任何字符，结果就算作匹配。

举个例子，比如想要匹配到元音字母，那可以这么写：

```
>>> re.search(r'[aeiou]', 'I love 123 FishC.com!')
<_sre.SRE_Match object; span=(3, 4), match='o'>
```

有些读者朋友可能会有疑惑："大写字母 I 也是元音字母，怎么不匹配它呢？"

这是因为正则表达式默认是区分大小写的，所以大写的 I 跟小写的 i 会区分开。解决的方案有两种：第一是关闭大小写敏感模式，这个后边再讲；第二就是修改的字符类：

```
>>> re.search(r'[aeiouAEIOU]', 'I love 123 FishC.com!')
<_sre.SRE_Match object; span=(0, 1), match='I'>
```

如上，字符类中的任何一个字符匹配，那么就算匹配成功。在中括号中，还可以使用小横杠来表示范围：

```
>>> re.search(r'[a-z]', 'I love 123 FishC.com!')
<_sre.SRE_Match object; span=(2, 3), match='l'>
```

同样可以用来表示数字的范围：

```
>>> re.search(r'[0-2][0-5][0-5]', 'I love 123 FishC.com!')
<_sre.SRE_Match object; span=(7, 10), match='123'>
```

### 14.5.5 重复匹配

数字范围的问题解决了。接下来需要处理第二个问题——匹配个数的问题。用大括号这对元字符来实现重复匹配的功能：

```
>>> re.search(r'ab{3}c', 'abbbc')
<_sre.SRE_Match object; span=(0, 5), match='abbbc'>
```

看，不多不少，正好三个可以匹配。

如果有五个？那不好意思，匹配不了：

```
>>> re.search(r'ab{3}c', 'abbbbbc')
>>>
```

重复的次数也可以取一个范围：

```
>>> re.search(r'ab{3,5}c', 'abbbbbc')
<_sre.SRE_Match object; span=(0, 7), match='abbbbbc'>
>>> re.search(r'ab{3,5}c', 'abbbc')
<_sre.SRE_Match object; span=(0, 5), match='abbbc'>
```

嗯，看到大家似乎已经信心满满、跃跃欲试，我忍不住还是要来打击一下大家：请问如何用正则表达式匹配 0～255 这个范围的数？

我知道有些读者朋友想都不用想就会这么写：

```
>>> re.search(r'[0-255]', '188')
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

或者会这么写：

```
>>> re.search(r'[0-2][0-5][0-5]', '188')
>>>
```

怎么样？果然跟你想象的不一样吧！

要记住，正则表达式匹配的字符串，所以数字对于字符来说只有 0～9，像 123 就是由'1''2''3'三个字符构成的。[0-255]这个字符类表示 0～2 还有两个 5，所以是匹配 0125 四个数字中任何一个。

要匹配 0～255 这个范围的数字，正则表达式应该这么写：

```
>>> re.search(r'[0-1]\d\d|2[0-4]\d|25[0-5]', '188')
<_sre.SRE_Match object; span=(0, 3), match='188'>
```

来试试匹配 ip 地址：

```
>>> re.search(r'(([01]\d\d|2[0-4]\d|25[0-5]\.){3}([01]\d\d|2[0-4]\d|25[0-5])', 'other192.
168.1.1other')
>>>
```

小括号是表示分组，这跟数学中小括号起到的作用类似。一个小组就是一个整体，我们后边加上重复次数{3}，表示这个小组的规则需要重复匹配三次才算成功。

那现在问题出在哪儿呢？眼尖的朋友发现了——这里没有充分考虑数字的位数。

因为数字 1 并不会刻意写成 001 这样三位数，所以再稍作修改：

```
>>> re.search(r'(([01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])\.){3}([01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])', 'other192.168.1.1other')
<_sre.SRE_Match object; span=(5, 16), match='192.168.1.1'>
```

搞定！大家现在应该可以充分理解"当你发现一个问题可以用正则表达式来解决的时候，于是你就有两个问题了"这句话。但大家也充分了解到掌握正则表达式的必要性。由于这里主要是讲解 Python 的爬虫应用，所以并没有花太多的时间来讲解正则表达式的隐藏技能。

这里作者翻译了 Python 官方的一篇关于正则表达式的 HOWTO 文档，大家可以参考：http://bbs.fishc.com/thread-57073-1-1.html。

### 14.5.6 特殊符号及用法

在 Python 中，正则表达式也是以字符串的形式来描述的。正则表达式的强大之处在于特殊符号的应用，特殊符号定义了字符集合、子组匹配、模式重复次数。正是这些特殊符号使得一个正则表达式可以匹配一个复杂的规则而不仅仅只是一个字符串。

表 14-1 说明了 Python3 正则表达式特殊符号及用法。

**表 14-1 Python3 正则表达式特殊符号及用法**

| 字　符 | 含　义 |
| --- | --- |
| . | 表示匹配除了换行符外的任何字符。注：通过设置 re.DOTALL 标志可以使.匹配任何字符(包含换行符) |
| \| | A \| B，表示匹配正则表达式 A 或者 B |
| ^ | (脱字符)匹配输入字符串的开始位置。如果设置了 re.MULTILINE 标志，^也匹配换行符之后的位置 |
| $ | 匹配输入字符串的结束位置。如果设置了 re.MULTILINE 标志，$ 也匹配换行符之前的位置 |
| \ | 将一个普通字符变成特殊字符，例如，\d 表示匹配所有十进制数字。解除元字符的特殊功能，例如，\. 表示匹配点号本身。引用序号对应的子组所匹配的字符串 |
| [...] | 字符类，匹配所包含的任意一个字符。注：连字符-如果出现在字符串中间表示字符范围描述；如果出现在首位则仅作为普通字符。特殊字符仅有反斜线\保持特殊含义，用于转义字符。其他特殊字符如 *、+、? 等均作为普通字符匹配。脱字符^如果出现在首位则表示匹配不包含其中的任意字符；如果^出现在字符串中间就仅作为普通字符匹配 |
| {M,N} | M 和 N 均为非负整数，其中 M <= N，表示前边的 RE 匹配 M ～ N 次。注：{M,}表示至少匹配 M 次；{,N}等价于{0,N}；{N}表示需要匹配 N 次 |
| * | 匹配前面的子表达式零次或多次，等价于{0,} |
| + | 匹配前面的子表达式一次或多次，等价于{1,} |
| ? | 匹配前面的子表达式零次或一次，等价于{0,1} |
| *?, +?, ?? | 默认情况下 *、+和? 的匹配模式是贪婪模式(即会尽可能多地匹配符合规则的字符串)；*?、+? 和?? 表示启用对应的非贪婪模式。举个例子：对于字符串"FishCCC"，正则表达式 FishC+会匹配整个字符串，而 FishC+? 则匹配"FishC" |
| {M,N}? | 同上，启用非贪婪模式，即只匹配 M 次 |

续表

| 字　　符 | 含　　义 |
| --- | --- |
| (...) | 匹配圆括号中的正则表达式，或者指定一个子组的开始和结束位置<br>注：子组的内容可以在匹配之后被\数字再次引用<br>举个例子：(\w+) \1 可以字符串"FishC FishC. com"中的"FishC FishC"(注意有空格) |
| (?...) | (? 开头的表示为正则表达式的扩展语法(下边这些是 Python 支持的所有扩展语法) |
| (? aiLmsux) | 1. (? 后可以紧跟着 'a'、'i'、'L'、'm'、's'、'u'、'x'中的一个或多个字符，只能在正则表达式的开头使用<br>2. 每一个字符对应一种匹配标志：re-A(只匹配 ASCII 字符)，re-I(忽略大小写)，re-L(区域设置)，re-M(多行模式)，re-S(. 匹配任何符号)，re-X(详细表达式)，包含这些字符将会影响整个正则表达式的规则<br>3. 当你不想通过 re. compile()设置正则表达式标志，这种方法就非常有用啦<br>注意，由于(? x)决定正则表达式如何被解析，所以它应该总是被放在最前边(最多允许前边有空白符)。如果(? x)的前边是非空白字符，那么(? x)就发挥不了作用了 |
| (?:...) | 非捕获组，即该子组匹配的字符串无法从后边获取 |
| (?P<name>...) | 命名组，通过组的名字(name)即可访问到子组匹配的字符串 |
| (?P=name) | 反向引用一个命名组，它匹配指定命名组匹配的任何内容 |
| (?#...) | 注释，括号中的内容将被忽略 |
| (?=...) | 前向肯定断言。如果当前包含的正则表达式(这里以...表示)在当前位置成功匹配，则代表成功，否则失败。一旦该部分正则表达式被匹配引擎尝试过，就不会继续进行匹配了；剩下的模式在此断言开始的地方继续尝试。举个例子：love(? =FishC)只匹配后边紧跟着"FishC"的字符串"love" |
| (?!...) | 前向否定断言。这跟前向肯定断言相反(不匹配则表示成功，匹配表示失败)。<br>举个例子：FishC(?! \. com)只匹配后边不是". com"的字符串"FishC" |
| (?<=...) | 后向肯定断言。跟前向肯定断言一样，只是方向相反。<br>举个例子：(? <=love)FishC 只匹配前边紧跟着"love"的字符串"FishC" |
| (?<!...) | 后向否定断言。跟前向肯定断言一样，只是方向相反。<br>举个例子：(? <! FishC)\. com 只匹配前边不是"FishC"的字符串". com" |
| (? (id/name)yes-pattern\|no-pattern) | 1. 如果子组的序号或名字存在的话，则尝试 yes-pattern 匹配模式；否则尝试 no-pattern 匹配模式<br>2. no-pattern 是可选的<br>举个例子：(<)? (\w+@\w+(?:\. \w+)+)(? (1)>\|$)是一个匹配邮件格式的正则表达式，可以匹配<user@fishc. com>和 'user@fishc. com'，但是不会匹配 '<user@fishc. com' 或 'user@fishc. com>' |
| \ | 下边列举了由字符 '\' 和另一个字符组成的特殊含义。注意，'\' +元字符的组合可以解除元字符的特殊功能 |
| \序号 | 1. 引用序号对应的子组所匹配的字符串，子组的序号从 1 开始计算<br>2. 如果序号是以 0 开头，或者 3 个数字的长度。那么不会被用于引用对应的子组，而是用于匹配八进制数字所表示的 ASCII 码值对应的字符<br>举个例子：(. +) \1 会匹配"FishC FishC"或"55 55"，但不会匹配"FishCFishC"(注意，因为子组后边还有一个空格) |
| \A | 匹配输入字符串的开始位置 |
| \Z | 匹配输入字符串的结束位置 |
| \b | 匹配一个单词边界，单词被定义为 Unidcode 的字母数字或下横线字符<br>举个例子：\bFishC\b 会匹配字符串"love FishC"、FishC. "或"(FishC)" |

续表

| 字　　符 | 含　　义 |
| --- | --- |
| \B | 匹配非单词边界，其实就是与\b 相反<br>举个例子：py\B 会匹配字符串"python"、"py3"或"py2"，但不会匹配"py"、"py."或"py!" |
| \d | 1. 对于 Unicode(str 类型)模式：匹配任何一个数字，包括[0-9]和其他数字字符；如果开启了 re.ASCII 标志，就只匹配[0-9]<br>2. 对于 8 位(bytes 类型)模式：匹配[0-9]中任何一个数字 |
| \D | 匹配任何非 Unicode 的数字，其实就是与\d 相反；如果开启了 re.ASCII 标志，则相当于匹配[^0-9] |
| \s | 1. 对于 Unicode(str 类型)模式：匹配 Unicode 中的空白字符(包括[ \t\n\r\f\v]以及其他空白字符)；如果开启了 re.ASCII 标志，就只匹配[ \t\n\r\f\v]<br>2. 对于 8 位(bytes 类型)模式：匹配 ASCII 中定义的空白字符，即[ \t\n\r\f\v] |
| \S | 匹配任何非 Unicode 中的空白字符，其实就是与\s 相反；如果开启了 re.ASCII 标志，则相当于匹配[^\t\n\r\f\v] |
| \w | 1. 对于 Unicode(str 类型)模式：匹配任何 Unicode 的单词字符，基本上所有语言的字符都可以匹配，当然也包括数字和下横线；如果开启了 re.ASCII 标志，就只匹配[a-zA-Z0-9_]<br>2. 对于 8 位(bytes 类型)模式：匹配 ASCII 中定义的字母数字，即[a-zA-Z0-9_] |
| \W | 匹配任何非 Unicode 的单词字符，其实就是与\w 相反；如果开启了 re.ASCII 标志，则相当于[^a-zA-Z0-9_] |
| 转义符号 | 正则表达式还支持大部分 Python 字符串的转义符号：\a,\b,\f,\n,\r,\t,\u,\U,\v,\x,\\。<br>注：\b 通常用于匹配一个单词边界，只有在字符类中才表示“退格”；\u 和\U 只有在 Unicode 模式下才会被识别；八进制转义(\数字)是有限制的，如果第一个数字是 0，或者如果有 3 个八进制数字，那么就被认为是八进制数；其他情况则被认为是子组引用；至于字符串，八进制转义总是最多只能是 3 个数字的长度 |

有些读者看到这个内心可能会犯嘀咕：“好歹我也是见过世面的人啊，为了查找一个字符串，真的有必要掌握这么多新规则吗？”

实话说，不需要！这里只是帮大家把 Python3 所有支持的正则表达式语法给列举出来，实际应用只需要用到这里边的一小部分。另外的一大部分主要是为了应对“突发情况”而准备的。这个列表大家可以收藏起来，在需要的时候翻出来查一查就可以了，千万不要去死记硬背。

我们说的特殊符号其实是由两部分组成：一部分是元字符，另一部分是由反斜杠加上普通符号组成的(这有点像 Python 字符串的转义符)。

## 14.5.7　元字符

以下是正则表达式所有的元字符：

```
.  ^  $  *  +  ?  { }  [ ]  \  |  ( )
```

它们各自都有特殊的含义，例如点号(.)表示匹配除换行符外的任何字符，管道符(|)则有点类似于这个逻辑或操作：

```
>>> re.search(r"Fish(C|D)", "FishC")
<_sre.SRE_Match object; span=(0, 5), match='FishC'>
>>> re.search(r"Fish(C|D)", "FishD")
```

```
<_sre.SRE_Match object; span=(0, 5), match='FishD'>
```

脱字符(^)表示匹配字符串的开始位置，也就是说，只有目标字符串出现在开头才会匹配：

```
>>> re.search(r"^FishC", "I love FishC.com!")
>>> re.search(r"^FishC", "FishC.com!")
<_sre.SRE_Match object; span=(0, 5), match='FishC'>
```

美元符号($)则表示匹配字符串的结束位置，也就是说，只有目标字符串出现在末尾才会匹配：

```
>>> re.search(r"FishC$", "FishC.com!")
>>> re.search(r"FishC$", "love FishC")
<_sre.SRE_Match object; span=(5, 10), match='FishC'>
```

反斜杠在正则表达式中用处最为广泛，它既可以将一个普通字符变成特殊字符(这个待会儿讲)，它还可以解除元字符的特殊功能，如果反斜杠后边加上的是数字，那它还有两种用法：如果跟着的数字是1～99，那么它表示引用序号对应的子组所匹配的字符串；如果跟着的数字是0开头或者是三位数字，那么它是一个八进制数，表示的是这个八进制数对应的ASCII字符。

听到这里大家肯定是一头雾水，你先别急，我一步步给你解释。首先，小括号(( ))本身是一对元字符，被它们括起来的正则表达式称为一个子组。子组有什么用呢？变成子组的话，就可以把它当作一个整体，例如在后边对它进行引用：

```
>>> re.search(r"(FishC)\1", "FishC.com")
```

这里的\1表示引用前边序号为1的子组(也就是第一个子组)，所以(r"(FishC)\1"相当于r"FishCFishC"。因此你无法匹配只有一个"FishC"的"FishC.com"，要写连续两个"FishC"才能成功匹配：

```
>>> re.search(r"(FishC)\1", "FishCFishC")
<_sre.SRE_Match object; span=(0, 10), match='FishCFishC'>
```

如果反斜杠后边跟着的数字是0开头或者三位的数字，那么会把这三位数字作为一个八进制数来看待：

```
>>> re.search(r"(FishC)\060", "FishCFishC0")
<_sre.SRE_Match object; span=(5, 11), match='FishC0'>
>>> # 注：八进制 60 对应的 ASCII 码是数字 0
>>> re.search(r"\141FishC", "aFishCFishC")
<_sre.SRE_Match object; span=(0, 6), match='aFishC'>
>>> # 注：八进制 141 对应的 ASCII 码是小写字母 a
```

接下来是中括号([ ])这对元字符，说它是生成一个字符类，事实上就是一个字符集合。被它包围在里边的元字符都失去了特殊的功能，就像反斜杠加上元字符的作用是一样的：

```
>>> re.search(r"[.]", "FishC.com")
<_sre.SRE_Match object; span=(5, 6), match='.'>
```

字符类的意思就是在它里边的内容，都把它们当成普通字符类看待，除了几个特殊的字符：

(1) 小横杠(-),用它来表示范围:

```
>>> re.findall(r"[a-z]", "FishC.com")
['i', 's', 'h', 'c', 'o', 'm']
>>> # findall()表示找出所有匹配的内容,并将结果返回为一个列表。
```

(2) 反斜杠,反斜杠这里用于字符串转义,例如\n 表示匹配换行符号:

```
>>> re.search(r"[\n]", "FishC.com\n")
<_sre.SRE_Match object; span=(9, 10), match='\n'>
```

(3) 脱字符(^),它用于表示取反的意思:

```
>>> re.findall(r"[^a-z]", "FishC.com")
['F', 'C', '.']
```

最后介绍的元字符是用来做重复的事情,例如前面讲的大括号({ }):

```
>>> re.search(r"FishC{3}", "FishCCC.com")
<_sre.SRE_Match object; span=(0, 7), match='FishCCC'>
```

如果前边是一个子组,那么表示整个子组重复的次数:

```
>>> re.search(r"(FishC){3}", "FishCCC.com")
>>> re.search(r"(FishC){3}", "FishCFishCFishC")
<_sre.SRE_Match object; span=(0, 15), match='FishCFishCFishC'>
```

另外还可以表示一个范围,就是多少次到多少次之间:

```
>>> re.search(r"(FishC){1,3}", "FishCFishCFishC")
<_sre.SRE_Match object; span=(0, 15), match='FishCFishCFishC'>
```

这里有一点需要注意,在正则表达式中,你不能随便使用空格:

```
>>> re.search(r"(FishC){1, 3}", "FishCFishCFishC")
>>>
```

你看,加上空格它就匹配不了了。

另外表示重复的元字符还有:*,+和?

- 星号(*)相当于{0,}。
- 加号(+)相当于{1,}。
- 问号(?)相当于{0,1}。

如果条件一样,推荐大家使用*、+和?,因为第一它们更加简洁,第二就是正则表达式引擎内部对这三个符号进行了优化,所以效率要比使用大括号高一些。

## 14.5.8 贪婪和非贪婪

关于重复的操作,有一点需要注意的,就是正则表达式默认是启用贪婪的匹配方式。什么是贪婪的匹配方式?就是说,只要在符合的条件下,它会尽量多地去匹配:

```
>>> s = "<html><title>I love FishC.com</title></html>"
>>> re.search('<.+>', s)
```

```
<_sre.SRE_Match object; span = (0, 44), match = '<html><title>I love FishC.com</title></html>'>
```

上面的代码,本来想匹配<html>,但这里由于贪婪模式的原因,它直接匹配了整个字符串。很明显这不是我们想要的。我们希望在遇到第一个“>”的时候就停下来,需要使用非贪婪模式。那非贪婪模式怎么启用呢?很简单,在表示重复的元字符后边再加上一个问号(?)即可:

```
>>> re.search('<.+?>', s)
<_sre.SRE_Match object; span = (0, 6), match = '<html>'>
```

好啦,正则表达式的所有元字符终于全部介绍完毕了。

## 14.5.9　反斜杠+普通字母=特殊含义

正则表达式的特殊符号除了元字符外,还有一种就是通过反斜杠加上普通字母构成的特殊符号。

首先是反斜杠加序号(\序号):

(1) 如果这个序号的范围是 1～99,那么表示引用序号对应的子组所匹配的字符串(子组的序号是从 1 开始算起的);

(2) 如果序号是以 0 开头,或者是三位数字的长度。那么不会被用于引用对应的子组,而是用于匹配八进制数字所表示的 ASCII 码值对应的字符。

\A 跟脱字符(^)在默认情况下是一样的,都表示匹配字符串的起始位置。也就是说,只要前边是\A 或者^符号,那么这个字符就必须出现在字符串的开头才算匹配。

\Z 跟美元符号 $ 在默认情况下是一样的,都表示匹配字符串的结束位置。

注意,刚才介绍的是在默认情况下一样,并不是说它们完全一样。因为正则表达式还有个标志的设置,如果设置了 re.MULTILINE 标志,那么^和 $ 元字符还可以匹配换行符的位置,而\A 和\Z 则只能匹配字符串的起始和结束位置。

这些匹配位置的字符都有一个名字:零宽断言,言下之意就是它们不会匹配任何字符,它们只用于匹配一个位置。

接下来是\b,它也是一个零宽断言,表示匹配一个单词的边界,单词这里被定义为 Unidcode 的字母数字或下横线字符。举个例子:

```
>>> re.findall(r"\bFishC\b", "FishC.com!FishC_com!FishC (FishC)")
['FishC', 'FishC', 'FishC']
```

注意,这里下横线是被定义为“单词”,所以字符串"FishC_com"是不会被匹配的:

```
>>> re.search(r"\bFishC\b", "FishC_com")
```

与\b 相反,\B 匹配的则是非单词边界。

还有\d 匹配的是 Unicode 中定义的数字字符,Unicode 是 Python3 默认的字符串类型。如果开启了 re.ASCII 标志,表示匹配 ASCII 码中定义的数字,也就是 0～9。如果你在字符串前加上 b,说明你想将字符串定义为 bytes 类型,那么匹配的就是 0～9。

与\d 相反,\D 匹配的是非 Unicode 定义的数字字符。

同样,\s 表示匹配任何空白字符,例如\t 表示 tab 键(制表键),\n 表示换行符,\r 表示回

车,\f 表示换页符,\v 则表示垂直的 tab 键(\t 是水平制表键)。

同理,\S 是\s 的取反。

\w 表示匹配 Unicode 中定义的单词字符,如果开启了 re.ASCII 标志,则只匹配[a-zA-Z0-9_];否则,每个字都属于单词字符的范围:

```
>>> re.findall(r"\w", "我爱鱼C工作室(love_FishC.com!)")
['我', '爱', '鱼', 'C', '工', '作', '室', 'l', 'o', 'v', 'e', '_', 'F', 'i', 's', 'h', 'C', 'c', 'o', 'm']
```

同样,\W 是\w 的取反。

除此之外,正则表达式还支持大部分 Python 字符串的转义符号:\a,\b,\f,\n,\r,\t,\u,\U,\v,\x,\\。

注 1:\b 通常用于匹配一个单词边界,只有在字符类中才表示"退格"。

注 2:\u 和\U 只有在 Unicode 模式下才会被识别。

注 3:八进制转义(\数字)是有限制的,如果第一个数字是 0,或者如果有三位八进制数字,那么就被认为是八进制数;其他情况则被认为是子组引用;至于字符串,八进制转义总是最多为三位数字的长度。

## 14.5.10 编译正则表达式

如果需要重复使用某个正则表达式,那么可以先将该正则表达式编译成模式对象。使用 re.compile()方法来进行编译:

```
>>> p = re.compile("[A-Z]")
>>> p.search("I love FishC.com!")
<_sre.SRE_Match object; span=(0, 1), match='I'>
>>> p.findall("I love FishC.com!")
['I', 'F', 'C']
```

正如大家所见,使用的方法跟调用模块级别的方法名是一样的,例如 search()和 findall()。不过第一参数就不再需要了,只需要传入待匹配的字符串即可。

## 14.5.11 编译标志

通过编译标志,可以修改正则表达式的工作方式。表 14-2 列举了可以使用的编译标志。

**表 14-2 编译标志**

| 标 志 | 含 义 |
|---|---|
| ASCII, A | 使得转义符号如\w,\b,\s 和\d 只能匹配 ASCII 字符 |
| DOTALL, S | 使得.匹配任何符号,包括换行符 |
| IGNORECASE, I | 匹配的时候不区分大小写 |
| LOCALE, L | 支持当前的语言(区域)设置 |
| MULTILINE, M | 多行匹配,影响^和 $ |
| VERBOSE, X (for 'extended') | 启用详细的正则表达式 |

**A,ASCII**

使得\w、\W、\b、\B、\s 和\S 只匹配 ASCII 字符,而不匹配完整的 Unicode 字符。这个标

志仅对 Unicode 模式有意义，并忽略字节模式。

**S，DOTALL**

使得点号(.)可以匹配任何字符，包括换行符。如果不使用这个标志，点号(.)将匹配除了换行符的所有字符。

**I，IGNORECASE**

字符类和文本字符串在匹配的时候不区分大小写。举个例子，正则表达式[A-Z]也将会匹配对应的小写字母，像 FishC 可以匹配 FishC、fishc 或 FISHC 等。如果你不设置 LOCALE，则不会考虑语言(区域)设置这方面的大小写问题。

**L，LOCALE**

使得\w、\W、\b 和\B 依赖当前的语言(区域)环境，而不是 Unicode 数据库。

区域设置是 C 语言的一个功能，主要作用是消除不同语言之间的差异。例如，你正在处理的是法文文本，你想使用\w+来匹配单词，但是\w 只是匹配[A-Za-z]中的单词，并不会匹配'é'或'ç'。如果你的系统正确地设置了法语区域环境，那么 C 语言的函数就会告诉程序'é'或'ç'也应该被认为是一个字符。当编译正则表达式的时候设置了 LOCALE 的标志，\w+就可以识别法文了，但速度多少会受到影响。

**M，MULTILINE**

通常脱字符(^)只匹配字符串的开头，而美元符号($)则匹配字符串的结尾。当这个标志被设置的时候，^不仅匹配字符串的开头，还匹配每一行的行首；$ 不仅匹配字符串的结尾，还匹配每一行的行尾。

**X，VERBOSE**

这个标志使正则表达式可以写得更好看和更有条理，因为使用了这个标志，空格会被忽略(除了出现在字符类中和使用反斜杠转义的空格)；这个标志同时允许你在正则表达式字符串中使用注释，井号(#)后边的内容是注释，不会递交给匹配引擎(除了出现在字符类中和使用反斜杠转义的\#)。

下边是使用 re.VERBOSE 的例子，大家看下正则表达式的可读性是不是提高了：

```
charref = re.compile(r"""
&[#]                    # 开始数字引用
(
    0[0-7]+             # 八进制格式
   |[0-9]+              # 十进制格式
   |x[0-9a-fA-F]+       # 十六进制格式
)
;                       # 结尾分号
""", re.VERBOSE)
```

如果没有设置 VERBOSE 标志，那么同样的正则表达式会写成：

```
charref = re.compile("&#(0[0-7]+|[0-9]+|x[0-9a-fA-F]+);")
```

哪个可读性更佳？相信大家已经心里有底了。

### 14.5.12　实用的方法

首先说 search()方法，模块级别的 search()方法就是直接调用 re.search()，编

译后的正则表达式模式对象也同样拥有 search()方法。那请问：它们有区别吗？

下面，看看它们的原型：

```
re.search(pattern, string, flags = 0)
regex.search(string[, pos[, endpos]])
```

由于 flags 标志是在编译的时候就同时编译进去了，所以模式对象就不需要 flags 了。另外模式对象的 search()方法还可以设置搜索的开始和结束位置。

还有，re.search()方法并不会立刻返回你可以使用的字符串，取而代之是返回一个匹配对象。

```
>>> result = re.search(r" (\w+) (\w+)", "I love FishC.com!")
>>> result
<_sre.SRE_Match object; span = (1, 12), match = ' love FishC'>
```

这时候需要使用匹配对象的一些方法才能获得需要的内容。例如，使用 group()才可以获得匹配的字符串：

```
>>> result.group()
' love FishC'
```

值得一提的是，如果正则表达式中存在子组，那么子组会将匹配的内容进行捕获。通过在 group()中设置序号，可以提取到对应的子组捕获的内容：

```
>>> result.group(1)
'love'
>>> result.group(2)
'FishC'
```

然后 start()、end()和 span()分别返回匹配的开始位置、结束位置以及匹配的范围：

```
>>> result.start()
1
>>> result.end()
12
>>> result.span()
(1, 12)
```

接下来是 findall()方法。这个容易，fandall()方法不就是找到所有匹配的内容，然后把它们组织成列表返回吗？没错，这是在正则表达式里边没有子组的情况下做的事。如果正则表达式里边包含了子组，那么 findall()会更聪明。下边通过案例来讲解。这一次将唯美图片贴吧的一些图片下载下来吧。

目标 URL：http://tieba.baidu.com/p/3823765471

踩点结果如图 14-17 所示。

一轮踩点下来，我们发现该贴吧的图片都是包含在<img class="BDE_Image">标签中的，例如：

```
< img class = " BDE _ Image " src = " http://imgsrc. baidu. com/forum/w % 3D580/sign =
f9cf09409c25bc312b5d01906ede8de7/8f0ede0735fae6cdafb377ef0ab30f2443a70fda.jpg" pic_ext =
"jpeg" changedsize = "true" width = "560" height = "497">
```



图 14-17　踩点

其中,width 和 height 可能会出现在 class="BDE_Image"以及 src 之间。

因此,可以写出对应的正则表达式应该是:

```
r'< img class = "BDE_Image". * ?src = "[^"] * \. jpg". * ?>'
```

不妨先用 IDLE 测试下:

```
>>> import urllib. request
>>> import re
>>> response = urllib. request. urlopen("http://tieba. baidu. com/p/3823765471")
>>> html = response. read(). decode("utf - 8")
>>> p = r'< img class = "BDE_Image". * ?src = "[^"] * \. jpg". * ?>'
>>> imglist = re. findall(p, html)
>>> for each in imglist:
    print(each)

< img class = "BDE_Image" src = "http://imgsrc. baidu. com/forum/w % 3D580/sign = f9cf09409c25bc31-
2b5d01906ede8de7/8f0ede0735fae6cdafb377ef0ab30f2443a70fda. jpg" pic _ ext = " jpeg" changedsize =
"true" width = "560" height = "497">
< img class = "BDE_Image" src = "http://imgsrc. baidu. com/forum/w % 3D580/sign = 35c4709bb9315c60-
43956be7bdb0cbe6/cc223ffae6cd7b894b6be60d0a2442a7d8330eda. jpg" pic _ ext = " jpeg" changedsize =
"true" width = "560" height = "497">
…
```

看起来是成功了,那下一步要解决的问题就是如何把里边的地址提取出来? 我知道不少执行力比较强的读者已经开始动手了。

等等,你! 慢! 着!

这里有更好的方法:

```
p = r'< img class = "BDE_Image". * ?src = "([^"] * \. jpg)". * ?>'
```

其实就是将图片的地址用小括号分组,先看看是否能成功实现:

```
>>> p = r'< img class = "BDE_Image". * ?src = "([^"] * \. jpg)". * ?>'
>>> imglist = re. findall(p, html)
>>> for each in imglist:
    print(each)

http://imgsrc. baidu. com/forum/w % 3D580/sign = f9cf09409c25bc312b5d01906ede8de7/8f0ede0735fa-
```

```
e6cdafb377ef0ab30f2443a70fda.jpg
http://imgsrc.baidu.com/forum/w%3D580/sign=35c4709bb9315c6043956be7bdb0cbe6/cc223ffae6cd-
7b894b6be60d0a2442a7d8330eda.jpg
…
```

好了，现在告诉你为什么会如此方便。这是因为在 findall()方法中，如果给出的正则表达式包含了一个或者多个子组，就会返回子组中匹配的内容。如果存在多个子组，那么它还会将匹配的内容组合成元组的形式再返回。

最后把程序完善起来：

```
# p14_10.py
import urllib.request
import re
import os

def open_url(url):
    req = urllib.request.Request(url)
    req.add_header('User-Agent', 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/
537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36')
    page = urllib.request.urlopen(req)
    html = page.read().decode('utf-8')
    return html

def get_img(html):
    p = r'<img class="BDE_Image".*?src="([^"]*\.jpg)".*?>'
    imglist = re.findall(p, html)
    try:
        os.mkdir("NewPics")
    except FileExistsError:
        # 如果该文件夹已存在则覆盖保存!
        pass
    os.chdir("NewPics")
    for each in imglist:
        filename = each.split("/")[-1]
        urllib.request.urlretrieve(each, filename, None)

if __name__ == '__main__':
    url = "http://tieba.baidu.com/p/3823765471"
    get_img(open_url(url))
```

程序执行效果如图 14-18 所示。

看，用好了正则表达式是不是很方便、很神奇？不过 findall()方法有时候会让你感觉很疑惑，很迷茫。举个例子，拿出前边匹配 IP 的正则表达式，然后依法炮制来获取代理 IP 地址：

```
# p14_11.py
import urllib.request
import re

def open_url(url):
    req = urllib.request.Request(url)
    req.add_header('User-Agent', 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/
```



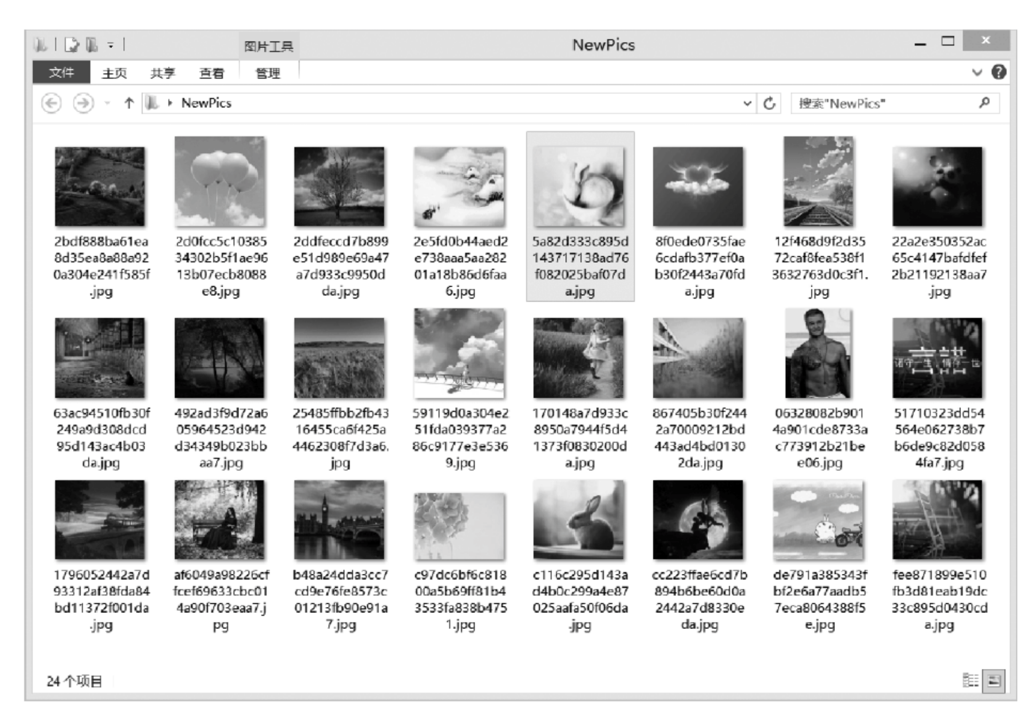



图 14-18　下载唯美图

```
537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36')
    page = urllib.request.urlopen(req)
    html = page.read().decode('utf - 8')
    return html

def get_img(html):
    p = r'(([01]{0,1}\d{0,1}\d|2[0 - 4]\d|25[0 - 5])\.){3}([01]{0,1}\d{0,1}\d|2[0 - 4]\d|
25[0 - 5])'
    imglist = re.findall(p, html)

    for each in imglist:
        print(each)

if __name__ == '__main__':
    url = "http://cn - proxy.com/"
    get_img(open_url(url))
```

程序执行后看到了奇怪的结果：

```
>>>
('57.', '57', '150')
('249.', '249', '126')
('59.', '59', '104')
...
```

怎么会这样呢？

这是因为在正则表达式中使用了三个子组，所以 findall()会自以为很聪明地帮我们把结果分好类，然后再返回给我们。它以为这样做我们会很开心，我们也只能呵呵了……

那有没有办法解决呢？答案是肯定有的！

要解决这个问题，可以让子组不捕获匹配的内容。需要使用到扩展语法：(?:…)，左小括号(()的后边紧跟着问号(?)，这时候问号(?)就不表示匹配前边内容零次或者多次了，因为它前边并不是表示重复的元字符。所以聪明的发明者把这样的组合认为是正则表达式的扩展语法。(?:…)表示非捕获组，即该子组匹配的字符串无法从后边获取。

姑且试试看吧，把正则表达式改成：

```
p = r'(?:(?:[01]{0,1}\d{0,1}\d|2[0-4]\d|25[0-5])\.){3}(?:[01]{0,1}\d{0,1}\d|2[0-4]\d|
25[0-5])'
```

看，真的成功了：

```
>>>
124.254.57.150
101.71.27.120
122.96.59.104
…
```

另外还有一些比较实用的方法，例如 finditer()方法会将结果返回给一个迭代器，这样方便你以迭代的方式获取；sub()方法是实现字符串的替换……还有一些特殊的语法，例如前向断言和后向断言，这些大家都可以在文档(注：http://bbs.fishc.com/thread-57073-1-1.html)中找到详细的介绍，这里就不再赘述了。

## 14.6 异常处理



高级语言的一个优秀特性就是提供了异常处理机制，让程序可以从容地处理每一个异常，不至于因为一个小错误而导致整个程序崩溃。大部分高级语言处理错误的方法都是通过检测异常、处理异常来实现的，当然 Python 也不例外。

用程序进行互联网访问的时候，出现异常是再正常不过的事情了。例如大家实现一个程序，通过几十个代理 IP 实现爬虫操作，如果其中一个代理 IP 突然不响应了，就会报错。这种错误触发率极高，全部代理 IP 都能用那才怪咧。但是一个出问题并不会影响到整个脚本的任务，所以捕获到此类异常的时候，直接忽略它即可。

### 14.6.1 URLError

当 urlopen 无法处理一个响应的时候，就会引发 URLError 异常。同时会伴随一个 reason 属性，用于包含一个由错误编码和错误信息组成的元组。

下面，我们稍微感受一下：

```
>>> import urllib
>>> req = urllib.request.Request('http://www.demo-fishc.com')
>>> try:
	urllib.request.urlopen(req)
except urllib.error.URLError as e:
	print(e.reason)
```

```
Bad Request
>>>
```

## 14.6.2 HTTPError

HTTPError 是 URLError 的子类，服务器上每一个 HTTP 的响应都包含一个数字的“状态码”。有时候状态码会指出服务器无法完成的请求类型，一般情况下 Python 会帮你处理一部分这类响应（例如，响应的是一个“重定向”，要求客户端从别的地址来获取文档，那么 urllib 会自动为你处理这个响应）；但是呢，有一些无法处理的，就会抛出 HTTPError 异常。这些异常包括典型的 404（页面无法找到）、403（请求禁止）和 401（验证请求）。

因为 Python 默认会自动帮你处理重定向方面的内容（状态码 300～399 范围），状态码 100～299 的范围是表示成功，所以你需要关注的是 400～599 这个范围的状态码（因为它们代表响应出了问题）。其中，出现 4xx 的状态码，说明问题来自客户端，就是你自己哪里做错了；出现 5xx 的状态码，那就表示与你无关了，是来自服务器的问题。

表 14-3 列举了常用的 HTTP 状态码以及详细的含义。

**表 14-3 常用 HTTP 状态码**

| 状态码 | 内 容 | 说 明 |
|---|---|---|
| | | |
| 1xx | | 这一类型的状态码，代表请求已被接受，需要继续处理 |
| 100 | Continue | 收到请求，客户端应当继续发送请求 |
| 101 | Switching Protocols | 服务器通过 Upgrade 消息头通知客户端采用不同的协议来完成这个请求 |
| | | |
| 2xx | 成功 | 这一类型的状态码，代表请求已成功被服务器接收、理解并接受 |
| 200 | OK | 请求已成功，请求的响应头或数据体将随此响应返回 |
| 201 | Created | 请求已经被实现，而且有一个新的资源已经依据请求的需要而创建，且其 URI 已经随 Location 头信息返回 |
| 202 | Accepted | 服务器已接受请求，但尚未处理。正如它可能被拒绝一样，最终该请求可能会也可能不会被执行 |
| 203 | Non-Authoritative Information | 服务器已成功处理了请求，但返回的实体头部元信息不是在原始服务器上有效地确定集合，而是来自本地或者第三方的复制 |
| 204 | No Content | 服务器成功处理了请求，但没有返回任何实体内容 |
| 205 | Reset Content | 服务器成功处理了请求，且没有返回任何内容。但是与 204 响应不同，返回此状态码的响应要求请求者重置文档视图 |
| 206 | Partial Content | 服务器已经成功处理了部分 GET 请求 |
| 3xx | 重定向 | 这类状态码代表需要客户端采取进一步的操作才能完成请求。通常，这些状态码用来重定向，后续的请求地址（重定向目标）在本次响应的 Location 域中指明 |
| 300 | Multiple Choices | 被请求的资源有一系列可供选择的回馈信息，每个都有自己特定的地址和浏览器驱动的商议信息。用户或浏览器能够自行选择一个首选的地址进行重定向 |
| 301 | Moved Permanently | 被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个 URI 之一 |

续表

| 状态码 | 内　　容 | 说　　明 |
| --- | --- | --- |
| 302 | Found | 请求的资源现在临时从不同的 URI 响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求 |
| 303 | See Other | 对应当前请求的响应可以在另一个 URI 上被找到，而且客户端应当采用 GET 的方式访问那个资源 |
| 304 | Not Modified | 如果客户端发送了一个带条件的 GET 请求且该请求已被允许，而文档的内容（自上次访问以来或者根据请求的条件）并没有改变，则服务器应当返回这个状态码 |
| 305 | Use Proxy | 被请求的资源必须通过指定的代理才能被访问。Location 域中将给出指定的代理所在的 URI 信息，接收者需要重复发送一个单独的请求，通过这个代理才能访问相应资源 |
| 307 | Temporary Redirect | 请求的资源现在临时从不同的 URI 响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求 |
| | | |
| 4xx | 客户端错误 | 这类的状态码代表了客户端看起来可能发生了错误，妨碍了服务器的处理 |
| 400 | Bad Request | 由于包含语法错误，当前请求无法被服务器理解 |
| 401 | Unauthorized | 当前请求需要用户验证 |
| 402 | Payment Required | 该状态码是为了将来可能的需求而预留的 |
| 403 | Forbidden | 服务器已经理解请求，但是拒绝执行它 |
| 404 | Not Found | 请求失败，请求的资源在服务器上找不到 |
| 405 | Method Not Allowed | 请求中指定的请求方法不能被用于请求相应的资源 |
| 406 | Not Acceptable | 请求的资源的内容特性无法满足请求头中的条件，因而无法生成响应实体 |
| 407 | Proxy Authentication Required | 与 401 状态码类似，只不过客户端必须在代理服务器上进行身份验证 |
| 408 | Request Timeout | 请求超时。客户端没有在服务器预备等待的时间内完成一个请求的发送 |
| 409 | Conflict | 由于和被请求的资源的当前状态之间存在冲突，请求无法完成 |
| 410 | Gone | 被请求的资源在服务器上已经不再可用，而且没有任何已知的转发地址 |
| 411 | Length Required | 服务器拒绝在没有定义 Content-Length 头的情况下接收请求 |
| 412 | Precondition Failed | 服务器在验证在请求的头字段中给出先决条件时，没能满足其中的一个或多个 |
| 413 | Request Entity Too Large | 服务器拒绝处理当前请求，因为该请求提交的实体数据大小超过了服务器愿意或者能够处理的范围 |
| 414 | Request-URI Too Long | 请求的 URI 长度超过了服务器能够解释的长度，因此服务器拒绝对该请求提供服务 |
| 415 | Unsupported Media Type | 对于当前请求的方法和所请求的资源，请求中提交的实体并不是服务器中所支持的格式，因此请求被拒绝 |
| 416 | Requested Range Not Satisfiable | 如果请求中包含了 Range 请求头，并且 Range 中指定的任何数据范围都与当前资源的可用范围不重合，同时请求中又没有定义 If-Range 请求头，那么服务器就应当返回 416 状态码 |
| 417 | Expectation Failed | 在请求头 Expect 中指定的预期内容无法被服务器满足，或者这个服务器是一个代理服务器，它有明显的证据证明在当前路由的下一个节点上，Expect 的内容无法被满足 |

续表

| 状态码 | 内　　容 | 说　　明 |
|---|---|---|
| 5xx | 服务器错误 | 这类状态码代表了服务器在处理请求的过程中有错误或者异常状态发生 |
| 500 | Internal Server Error | 服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理 |
| 501 | Not Implemented | 服务器不支持当前请求所需要的某个功能 |
| 502 | Bad Gateway | 作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应 |
| 503 | Service Unavailable | 由于临时的服务器维护或者过载，服务器当前无法处理请求 |
| 504 | Gateway Timeout | 作为网关或者代理工作的服务器尝试执行请求时，未能及时从上游服务器(URI 标识出的服务器，例如 HTTP、FTP、LDAP)或者辅助服务器(例如 DNS)收到响应 |
| 505 | HTTP Version Not Supported | 服务器不支持，或者拒绝支持在请求中使用的 HTTP 版本 |

当出现一个错误的时候，服务器返回一个 HTTP 错误号和一个错误页面。可以使用 HTTPError 实例作为页面返回的响应对象。它同样也是拥有像 read()、geturl()和 info()这类方法。

```
>>> req = urllib.request.Request('http://www.fishc.com/demo.html')
>>> try:
	urllib.request.urlopen(req)
except urllib.error.HTTPError as e:
	print(e.code)
	print(e.read())

404
b'<?xml version="1.0" encoding="ISO-8859-1"?>\n<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML
1.0 Strict//EN"\n
…
```

### 14.6.3 处理异常

处理 HTTPError 或 URLRrror 异常有两种方法。

第一种写法是像这样：

```
# p14_12.py
from urllib.request import Request, urlopen
from urllib.error import URLError, HTTPError

req = Request(someurl)
try:
    response = urlopen(req)
except HTTPError as e:
    print('The server couldn\'t fulfill the request.')
    print('Error code: ', e.code)
except URLError as e:
```

```
        print('We failed to reach a server.')
        print('Reason: ', e.reason)
else:
# everything is fine
```

这里需要注意的一点是：except HTTPError 必须在前边，因为它是 URLError 的子类。所以如果把 except URLError 放前边，就会把 HTTPError 的内容过滤掉了。

第二种写法是像这样：

```
# p14_13.py
from urllib.request import Request, urlopen
from urllib.error import URLError

req = Request(someurl)
try:
    response = urlopen(req)
except URLError as e:
    if hasattr(e, 'reason'):
        print('We failed to reach a server.')
        print('Reason: ', e.reason)
    elif hasattr(e, 'code'):
        print('The server couldn\'t fulfill the request.')
        print('Error code: ', e.code)
else:
# everything is fine
```

虽然两种写法都可以实现，但比较推荐第二种写法。

## 14.7 安装 Scrapy



说到 Python 爬虫，“大牛们”都会不约而同地提起“西瓜皮”(Scrapy)。因为 Scrapy 是一个为了爬取网站数据、提取结构性数据而编写的应用框架，可以应用在包括数据挖掘、信息处理或存储历史数据等一系列的程序中。

Scrapy 最初是为了页面抓取(更确切地说，是网络抓取)所设计的，也可以应用在获取 API 所返回的数据(例如 Amazon Associates Web Services)或者通用的网络爬虫。

由于 Scrapy 目前不支持 Python3，因此需要安装 Python2.7 来使用 Scrapy。不过不用担心，Python2.7 和 Python3 是可以共存的(如果出现“兼容性”问题，请查看：http://bbs.fishc.com/thread-58701-1-1.html)。

安装步骤(Windows 下所需要的安装包均已提供，详见“安装 Scrapy 所需要的软件.zip”)：

(1) 安装 Python2.7(32 位版本)，地址为 https://www.python.org/downloads/release/python-279/。

(2) 打开“运行”对话框，输入 cmd。执行以下命令，设置环境变量：

```
C:\Python27\python.exe C:\Python27\tools\Scripts\win_add2path.py
```

(3) 重新输入 cmd,执行命令 C:\Python27\python. exe--version,如果显示 Python2. 7. 9,则说明成功;如果没有显示,请服用 Windows“特效药”——重启系统尝试一下。

(4) 安装 pywin32(32 位版本),地址为 http://sourceforge. net/projects/pywin32/。

(5) 安装 pip,地址为 https://pip. pypa. io/en/latest/installing. html。

① 下载 get-pip. py。

② 进入 cmd,执行命令 python get-pip. py。

重要提示:如果你的用户名是中文,那么执行上面操作会报编码错误,请在 python 目录(Python27\Lib\site-packages)中新建一个文件 sitecustomize. py。

内容如下:

```
import sys
sys.setdefaultencoding('gb2312')
```

③ 检查 Python27\Scripts 中是否有 pip. exe 并设置 Python27\Scripts 到环境变量中。

④ 重启 cmd,输入命令“pip --version”,如果显示版本号,则说明成功;如果没有显示,请继续服用 Windows“特效药”——重启系统尝试一下。

pip 实际上就是 Python 的一个安装软件的工具,有了它就可以轻松便捷地安装各种 Python 的模块了。离我们的目标不远了,接下来还需要 lxml 和 pyOpenSSL。

(6) 虽然可以用 pip 安装 lxml,但如果你使用的是 Windows 系统,则建议不要,因为 lxml 有很多依赖的软件,其他系统都是自带的,但 Windows 没有,所以还是老老实实使用 lxml 专门为 Windows 提供的安装包来安装吧。

(7) 接下来使用 pip 来安装 pyOpenSSL,需要说明的是,OpenSSL 一般在其他系统也是有预安装的,除了 Windows……来,pip 走起:

```
C:\>pip install pyOpenSSL
```

重要提示:这里 pip 需要微软的 VS2008 的 C 语言编译器,所以如果没有安装 VS2008 或者你的 VS 版本太高,也是不行的。可以安装微软为 Python 开发的:VCForPython27. msi。

(8) 最后一步,使用 pip 安装我们的主角 Scrapy:

```
pip install Scrapy
```

(9) 启动小爬爬:

```
C:\>Scrapy:0: UserWarning: You do not have a working installation of the service_identity
module: 'No module named service_identity'. Please install it from <https://pyp
i. python.org/pypi/service_identity> and make sure all of its dependencies are sa
tisfied. Without the service_identity module and a recent enough pyOpenSSL to s
upport it, Twisted can perform only rudimentary TLS client hostname verification
. Many valid certificate/hostname mappings may be rejected.
```

(10) 虽然没有报错,但有一个提醒需要处理,即提示我们需要安装 service_identity,有了 pip,安装模块就太简单了:

```
C:\>pip install service_identity
```

(11) 搞定,如图 14-19 所示。

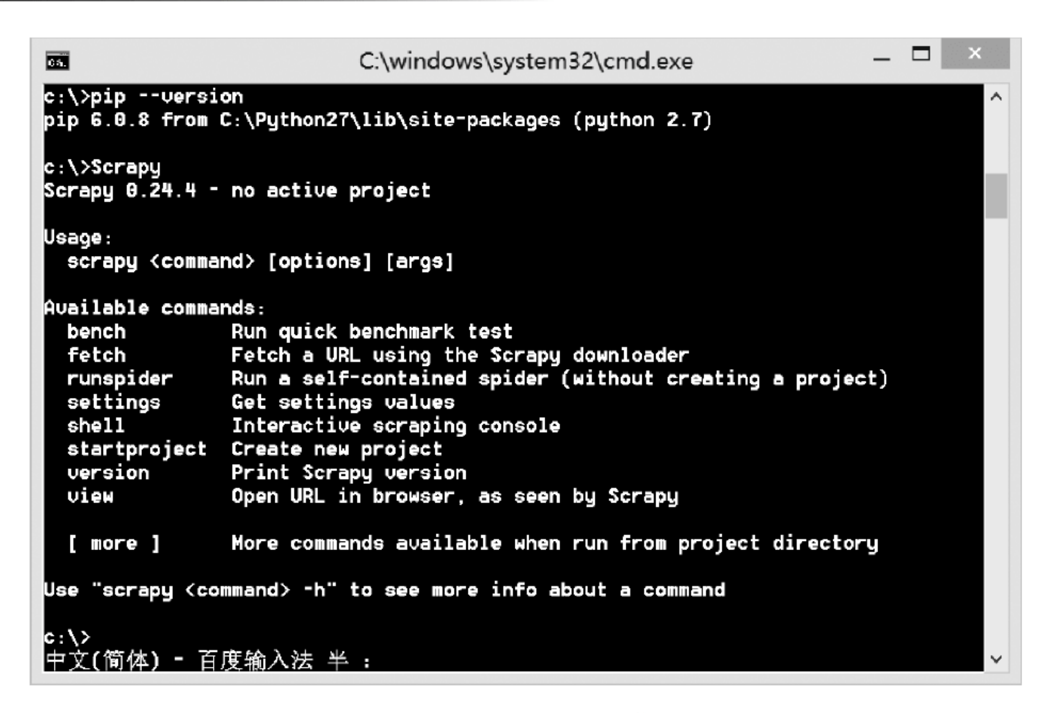



图 14-19　Scrapy 的安装

## 14.8　Scrapy 爬虫之初窥门径



有些读者可能会有疑惑："既然我们懂得了 Python 编写爬虫的技巧，那要这个所谓的爬虫框架又有什么用？"

其实懂得用 Python 写爬虫代码，就像你懂武功会打架，但行军打仗你不行，毕竟敌人是千军万马，纵使你再强，也只能是百人敌，要成为万人敌，你要学的就是排兵布阵，运筹帷幄。所以 Scrapy 就是 Python 爬虫的"孙子兵法"。

使用 Scrapy 抓取一个网站一共需要四个步骤：

（1）创建一个 Scrapy 项目；

（2）定义 Item 容器；

（3）编写爬虫；

（4）存储内容。

### 14.8.1　Scrapy 框架

学习怎么使用 Scrapy 之前，需要先来了解一下 Scrapy 的架构以及组件之间的交互。图 14-20 展现的是 Scrapy 的架构，包括组件及在系统中发生的数据流（图中箭头指示）。

#### 1. Scrapy Engine

Scrapy 引擎是爬虫工作的核心，负责控制数据流在系统中所有组件中流动，并在相应动作发生时触发事件。



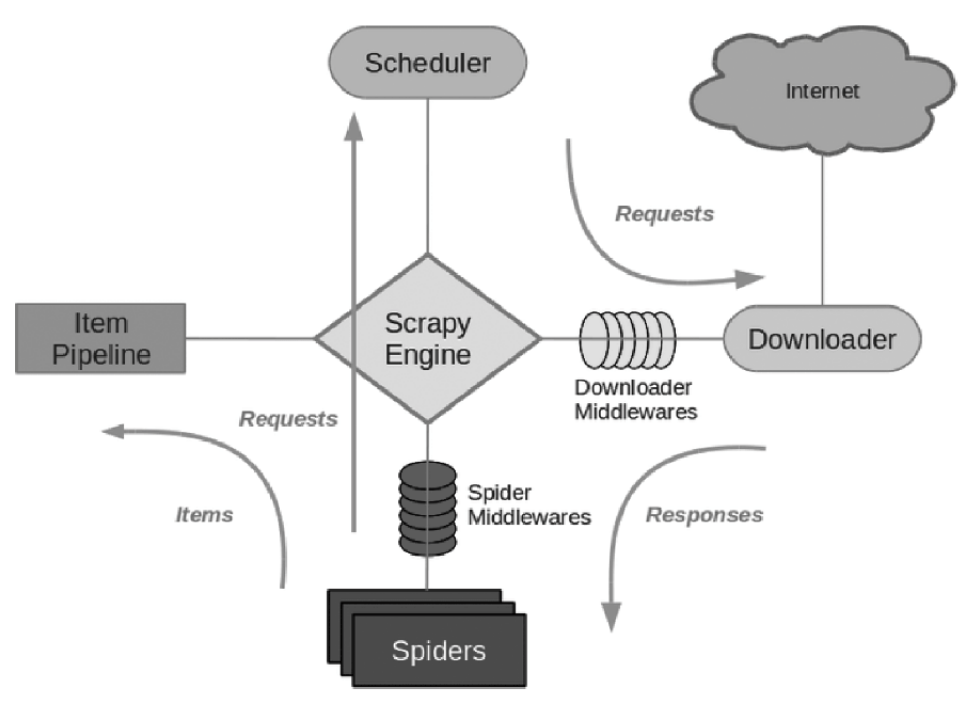



图 14-20　Scrapy 架构

**2. 调度器**

调度器(Scheduler)从引擎接受 request 并将它们入队,以便之后引擎请求它们时提供给引擎。

**3. 下载器**

下载器(Downloader)负责获取页面数据并提供给引擎,而后提供给 Spider。

**4. Spiders**

Spider 是 Scrapy 用户编写用于分析由下载器返回的 response,并提取出 item 和额外跟进的 URL 的类。

**5. Item Pipeline**

Item Pipeline 负责处理被 Spider 提取出来的 item。典型的处理有清理、验证及持久化(例如,存取到数据库中)。

接下来是两个中间件,它们用于提供一个简便的机制,通过插入自定义代码来扩展 Scrapy 的功能。

**6. 下载器中间件**

下载器中间件(Downloader middlewares)是在引擎及下载器之间的特定钩子(specific hook),处理 Downloader 传递给引擎的 response。

#### 7. Spider 中间件

Spider 中间件(Spider middlewares)是在引擎及 Spider 之间的特定钩子(specific hook),处理 spider 的输入(就是接收来自下载器的 response)和输出(就是发送 items 给 item pipeline 以及发送 requests 给调度器)。

下面给大家从头到尾演示一遍:

抓取 dmoz.org 网站有关 Python 的 Books(http://www.dmoz.org/Computers/Programming/Languages/Python/Books/)和 Resources(http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/)资源。

DMOZ 网站是一个著名的开放式分类目录(Open Directory Project),之所以称为开放式分类目录,是因为 DMOZ 不同于一般分类导航网站。DMOZ 中的所有内容都是由来自世界各地的志愿者共同维护与建设的,目前 DMOZ 是互联网上最大的、最广泛的人工目录。那就拿它来练练手,看 Scrapy 是否能胜任。

### 14.8.2 创建一个 Scrapy 项目

在开始爬取之前,需要先创建一个新的 Scrapy 项目,并进入打算存储代码的目录中,运行下列命令:

```
C:\ > scrapy startproject tutoria
```

该命令将会创建包含下列内容的 tutorial 目录:

```
tutorial/
    scrapy.cfg
    tutorial/
        __init__.py
        items.py
        pipelines.py
        settings.py
        spiders/
            __init__.py
            ...
```

这些文件构成了 Scrapy 爬虫框架。

- scrapy.cfg:项目的配置文件。
- tutorial/:该项目的 python 模块,之后将在此加入代码。
- tutorial/items.py:项目中的 item 文件。
- tutorial/pipelines.py:项目中的 pipelines 文件。
- tutorial/settings.py:项目的设置文件。
- tutorial/spiders/:放置 spider 代码的目录。

### 14.8.3 定义 Item 容器

Item 是保存爬取到的数据的容器,其使用方法和 python 字典类似,并且提供了额外保护



机制来避免因拼写错误导致的未定义字段错误。

首先根据需要从 dmoz.org 获取到的数据对 item 进行建模。例如需要从 dmoz 中获取名字、url 网址以及网站的描述。对此，需要在 item 中定义相应的字段。

编辑 tutorial 目录中的 items.py 文件：

```
import scrapy

class DmozItem(scrapy.Item):
    title = scrapy.Field()
    link = scrapy.Field()
    desc = scrapy.Field()
```

### 14.8.4 编写爬虫

接下来是编写爬虫类 Spider，Spider 是用户编写用于从网站上爬取数据的类。

其包含了一个用于下载的初始 URL，然后是如何跟进网页中的链接以及如何分析页面中的内容，还有提取生成 item 的方法。

创建一个自定义的 Spider 时，必须继承 scrapy.Spider 类，且定义以下三个属性：

- name——用于区别不同的 Spider。该名字必须是唯一的，不可以为不同的 Spider 设定相同的名字。
- start_urls——包含了 Spider 在启动时进行爬取的 url 列表。因此，第一个被获取到的页面将是其中之一。后续的 URL 则从初始的 URL 获取到的数据中提取。
- parse()——是 spider 的一个回调函数。当下载器返回 Response 的时候，该函数就会被调用，每个初始 URL 完成下载后生成的 Response 对象将会作为唯一的参数传递给该函数。该方法负责解析返回的数据(response data)，提取数据(生成 item)以及生成需要进一步处理的 URL 的 Request 对象。

以下的 Spider 代码保存在 tutorial/spiders 目录下的 dmoz_spider.py 文件中：

```
import scrapy

class DmozSpider(scrapy.Spider):
    name = "dmoz"
    allowed_domains = ["dmoz.org"]
    start_urls = [
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Books/",
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/"
    ]

    def parse(self, response):
        filename = response.url.split("/")[-2]
        with open(filename, 'wb') as f:
            f.write(response.body)
```

### 14.8.5 爬

通过命令行进入 tutorial 项目的根目录：

```
C:\ > cd tutorial
```

然后执行下列命令启动 spider：

```
C:\ > scrapy crawl dmoz
```

crawl dmoz 启动用于爬取 dmoz.org 的 spider，将得到类似的输出，如图 14-21 所示。

```
2015-02-28 23:31:37+0800 [scrapy] INFO: Scrapy 0.24.4 started (bot: tutorial)
2015-02-28 23:31:37+0800 [scrapy] INFO: Optional features available: ssl, http11

2015-02-28 23:31:37+0800 [scrapy] INFO: Overridden settings: {'NEWSPIDER_MODULE'
: 'tutorial.spiders', 'SPIDER_MODULES': ['tutorial.spiders'], 'BOT_NAME': 'tutor
ial'}
2015-02-28 23:31:37+0800 [scrapy] INFO: Enabled extensions: LogStats, TelnetCons
ole, CloseSpider, WebService, CoreStats, SpiderState
2015-02-28 23:31:37+0800 [scrapy] INFO: Enabled downloader middlewares: HttpAuth
Middleware, DownloadTimeoutMiddleware, UserAgentMiddleware, RetryMiddleware, Def
aultHeadersMiddleware, MetaRefreshMiddleware, HttpCompressionMiddleware, Redirec
tMiddleware, CookiesMiddleware, ChunkedTransferMiddleware, DownloaderStats
2015-02-28 23:31:37+0800 [scrapy] INFO: Enabled spider middlewares: HttpErrorMid
dleware, OffsiteMiddleware, RefererMiddleware, UrlLengthMiddleware, DepthMiddlew
are
2015-02-28 23:31:37+0800 [scrapy] INFO: Enabled item pipelines:
2015-02-28 23:31:37+0800 [dmoz] INFO: Spider opened
2015-02-28 23:31:37+0800 [dmoz] INFO: Crawled 0 pages (at 0 pages/min), scraped
0 items (at 0 items/min)
2015-02-28 23:31:37+0800 [scrapy] DEBUG: Telnet console listening on 127.0.0.1:6
023
2015-02-28 23:31:37+0800 [scrapy] DEBUG: Web service listening on 127.0.0.1:6080

2015-02-28 23:31:38+0800 [dmoz] DEBUG: Crawled (200) <GET http://www.dmoz.org/Co
mputers/Programming/Languages/Python/Resources/> (referer: None)
2015-02-28 23:31:38+0800 [dmoz] DEBUG: Crawled (200) <GET http://www.dmoz.org/Co
mputers/Programming/Languages/Python/Books/> (referer: None)
2015-02-28 23:31:38+0800 [dmoz] INFO: Closing spider (finished)
2015-02-28 23:31:38+0800 [dmoz] INFO: Dumping Scrapy stats:
        {'downloader/request_bytes': 516,
         'downloader/request_count': 2,
         'downloader/request_method_count/GET': 2,
         'downloader/response_bytes': 16342,
         'downloader/response_count': 2,
         'downloader/response_status_count/200': 2,
         'finish_reason': 'finished',
         'finish_time': datetime.datetime(2015, 2, 28, 15, 31, 38, 475000),
         'log_count/DEBUG': 4,
         'log_count/INFO': 7,
         'response_received_count': 2,
         'scheduler/dequeued': 2,
         'scheduler/dequeued/memory': 2,
         'scheduler/enqueued': 2,
         'scheduler/enqueued/memory': 2,
         'start_time': datetime.datetime(2015, 2, 28, 15, 31, 37, 481000)}
2015-02-28 23:31:38+0800 [dmoz] INFO: Spider closed (finished)
```

图 14-21　Scrapy 框架之初窥门径(一)

查看包含[dmoz]的输出，可以看到输出的日志中包含定义在 start_urls 的初始 URL，并且与 spider 中是一一对应的。在日志中可以看到其没有指向其他页面( (referer:None) )。

除此之外，更有趣的事情发生了。就像 parse()函数中指定的那样，有两个包含 URL 所对应的内容的文件被创建出来：Book 和 Resources，如图 14-22 所示。



| 名称 | 修改日期 | 类型 | 大小 |
| --- | --- | --- | --- |
| tutorial | 2/28, 星期六 23:31 | 文件夹 | |
| Books | 2/28, 星期六 23:31 | 文件 | 33 KB |
| Resources | 2/28, 星期六 23:31 | 文件 | 17 KB |
| scrapy.cfg | 2/28, 星期六 23:05 | CFG 文件 | 1 KB |

图 14-22　Scrapy 框架之初窥门径(二)

刚才发生了什么?

Scrapy 为 Spider 的 start_urls 属性中每个 URL 创建了 Request 对象,并将 parse()方法指定为回调函数。Request 对象经过调度,执行下载器并生成 Response 对象反馈回 Spider 类。

## 14.8.6　取

爬完整个网页,接下来就是取的过程了。大家还记得之前定义的 item 容器吧? 取就是这么一个大浪淘沙的过程——从得到的网页内容提取出我们需要的数据。

之前教大家是使用正则表达式,在 Scrapy 中是使用一种基于 XPath 和 CSS 的表达式机制:Scrapy Selectors。

Selector 是一个选择器,它有四个基本方法:

(1) xpath()——传入 xpath 表达式,返回该表达式所对应的所有节点的 selector list 列表。

(2) css()——传入 CSS 表达式,返回该表达式所对应的所有节点的 selector list 列表。

(3) extract()——序列化该节点为 unicode 字符串并返回 list。

(4) re()——根据传入的正则表达式对数据进行提取,返回 unicode 字符串 list 列表。

## 14.8.7　在 Shell 中尝试 Selector 选择器

为了介绍 Selector 的使用方法,接下来将要使用内置的 Scrapy shell。你需要先进入项目的根目录,执行下列命令来启动 Scrapy shell:

```
C:\ > scrapy shell
"http://www.dmoz.org/Computers/Programming/Languages/Python/Books/"
```

shell 的输出如图 14-23 所示。

在 Shell 载入后,你将获得 response 回应,存储在本地变量 response 中。

所以如果输入 response.body,你将会看到 response 的 body 部分,也就是抓取到的页面内容,如图 14-24 所示。

或者输入 response.headers 来查看它的 header 部分,如图 14-25 所示。

现在就像是一大堆沙子握在手里,里面有我们想要的金子,所以下一步就要用筛子把沙子去掉,淘出金子。selector 选择器就是这样一个筛子,正如刚才讲到的,你可以使用 response.selector.xpath()、response.selector.css()、response.selector.extract()和 response.selector.re()这四个基本方法。

```
2015-02-28 23:38:43+0800 [scrapy] INFO: Scrapy 0.24.4 started (bot: tutorial)
2015-02-28 23:38:43+0800 [scrapy] INFO: Optional features available: ssl, http11

2015-02-28 23:38:43+0800 [scrapy] INFO: Overridden settings: {'NEWSPIDER_MODULE'
: 'tutorial.spiders', 'SPIDER_MODULES': ['tutorial.spiders'], 'LOGSTATS_INTERVAL
': 0, 'BOT_NAME': 'tutorial'}
2015-02-28 23:38:43+0800 [scrapy] INFO: Enabled extensions: TelnetConsole, Close
Spider, WebService, CoreStats, SpiderState
2015-02-28 23:38:44+0800 [scrapy] INFO: Enabled downloader middlewares: HttpAuth
Middleware, DownloadTimeoutMiddleware, UserAgentMiddleware, RetryMiddleware, Def
aultHeadersMiddleware, MetaRefreshMiddleware, HttpCompressionMiddleware, Redirec
tMiddleware, CookiesMiddleware, ChunkedTransferMiddleware, DownloaderStats
2015-02-28 23:38:44+0800 [scrapy] INFO: Enabled spider middlewares: HttpErrorMid
dleware, OffsiteMiddleware, RefererMiddleware, UrlLengthMiddleware, DepthMiddlew
are
2015-02-28 23:38:44+0800 [scrapy] INFO: Enabled item pipelines:
2015-02-28 23:38:44+0800 [scrapy] DEBUG: Telnet console listening on 127.0.0.1:6
023
2015-02-28 23:38:44+0800 [scrapy] DEBUG: Web service listening on 127.0.0.1:6080

2015-02-28 23:38:44+0800 [dmoz] INFO: Spider opened
2015-02-28 23:38:45+0800 [dmoz] DEBUG: Crawled (200) <GET http://www.dmoz.org/Co
mputers/Programming/Languages/Python/Books/> (referer: None)
[s] Available Scrapy objects:
[s]   crawler    <scrapy.crawler.Crawler object at 0x03A78370>
[s]   item       {}
[s]   request    <GET http://www.dmoz.org/Computers/Programming/Languages/Python
/Books/>
[s]   response   <200 http://www.dmoz.org/Computers/Programming/Languages/Python
/Books/>
[s]   settings   <scrapy.settings.Settings object at 0x037417B0>
[s]   spider     <DmozSpider 'dmoz' at 0x3d75a70>
[s] Useful shortcuts:
[s]   shelp()           Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local objects
[s]   view(response)    View response in a browser

>>>
```

图 14-23　在 Shell 中尝试 Selector 选择器(一)

```
ort-abuse.dmoz.org"){A="utilities";y="report abuse";j="level-0";h=((D===0)&&(d==
="faq"))?"dmoz report abuse faq":"dmoz report abuse system";e=1}else{if(t
==="rdf.dmoz.org"){A="utilities";y="rdf";j="level-0";if((D===0)&&(d===" - ")){h=
"rdf - main"}else{if((D===1)&&(d===" - ")){h="rdf - file index"}else{if((D===1)&
&(d!==" - ")){h="rdf - "+d}}}e=1}}}if(e!==1){if((c==="cgi-bin")&&(t!=="search.dm
oz.org")){A="utilities";j="level-0";switch(d){case"apply":case"forgot":y="editor
s";h=(d==="forgot")?"editors - password reminder form":(s_265.getQueryParam("sub
mit").length===0)?"editors - application info":"editors - application";e=1;break
;case"add":case"update":case"update2":case"update3":case"reinstate":y="editors";
if(d==="add"){h="editors - submit a site instructions"}else{if(d==="update"){h="
editors - update listing instructions"}else{if(d==="update2"){h="editors - updat
e listing form"}else{if(d==="update3"){h="editors - update listing form received
"}else{if(d==="reinstate"){h="editors - account reinstatement form received"}}}}
}e=1;break;case"send":case"send2":y="editors";h=(d==="send")?"send editors feedb
ack":"editors feedback received";e=1;break}}if(e!==1){if((k==="desc.html")||(k=
=="faq.html")){w=(k==="desc.html")?"description":"faq";h=(D===1)?"branch categor
y ":(D===2)?"branch subcategory ":"branch level-"+D+" ";h=h+w}}s_265.trackExtern
alLinks=false;s_265.mmxgo=true;s_265.prop1=A;s_265.prop2=y;s_265.prop17=j;s_265.
pageName=h;s_265.t()}}())}var s_account="aoldmozodp,aolsvc";(function(){var b=do
cument,a=b.createElement("script");a.type="text/javascript";a.src=(location.prot
ocol==="https:"?"https://s":"http://o")+".aolcdn.com/omniunih.js";b.getElementsB
yTagName("head")[0].appendChild(a)}());\r\n</script>\r\n</div>\r\n</body>\r\n</h
tml>\r\n\r\n'
>>>
```

图 14-24　在 Shell 中尝试 Selector 选择器(二)

```
>>> response.headers
{'Cteonnt-Length': ['33399'], 'Content-Language': ['en'], 'Set-Cookie': ['JSESSI
ONID=00B4A690711C0099A4E4E5CEFE400EC6; Path=/'], 'Server': ['Apache'], 'Date': [
'Sat, 28 Feb 2015 15:41:14 GMT'], 'Content-Type': ['text/html;charset=UTF-8']}
>>>
```

图 14-25　在 Shell 中尝试 Selector 选择器(三)

## 14.8.8 使用 XPath

什么是 XPath？

XPath 是一门在网页中查找特定信息的语言。所以用 XPath 来筛选数据，要比使用正则表达式容易些。

下面是 XPath 表达式的例子及对应的含义：

- /html/head/title——选择 HTML 文档中<head>标签内的<title>元素。
- /html/head/title/text()——选择上面提到的<title>元素的文字。
- //td——选择所有的<td>元素。
- //div[@class="mine"]——选择所有具有 class="mine"属性的 div 元素。

上面仅仅是几个简单的 XPath 例子，实际上 XPath 要强大得多。如果你想了解更多关于 XPath 的内容，推荐学习这篇文章 http://www.w3school.com.cn/xpath/。

值得一提的是，response.xpath()、response.css()已经被映射到 response.selector.xpath()、response.selector.css()，所以直接使用 response.xpath()即可，如图 14-26 所示。

```
>>> response.xpath('//title')
[<Selector xpath='//title' data=u'<title>DMOZ - Computers: Programming: La'>]
>>> response.xpath('//title').extract()
[u'<title>DMOZ - Computers: Programming: Languages: Python: Books</title>']
>>> response.xpath('//title/text()')
[<Selector xpath='//title/text()' data=u'DMOZ - Computers: Programming: Language
s'>]
>>> response.xpath('//title/text()').extract()
[u'DMOZ - Computers: Programming: Languages: Python: Books']
>>> response.xpath('//title/text()').re('(\w+):')
[u'Computers', u'Programming', u'Languages', u'Python']
>>>
```

图 14-26　使用 XPath

## 14.8.9 提取数据

接下来尝试从这些页面中提取一些有用的数据。当然你可以通过输入 response.body 来观察 HTML 源码并确定 XPath 表达式。不过这里不建议你这么做，因为这样做太麻烦了。你完全可以利用谷歌浏览器的“审核元素”功能来观察(就像以前踩点的时候所做的一样)。

根据 item 中定义的，需要找到(书的)名字、URL 网址以及网站的描述。要找出它们的规律，然后通过 XPath 的语法把它们筛选出来，如图 14-27 所示。

发现需要的东西都在<ul></ul>标签中，每对<li></li>标签包含一组我们需要的信息。因此，可以用如下代码来捕获这个<li>标签(shell 根据 response 的类型自动为我们初始化了变量 sel，可以直接使用)：

```
sel.xpath('//ul/li')
```

从<li>标签中，可以这样捕获网站的描述，并用列表返回：

```
sel.xpath('//ul/li/text()').extract()
```

可以这样捕获网站的标题：

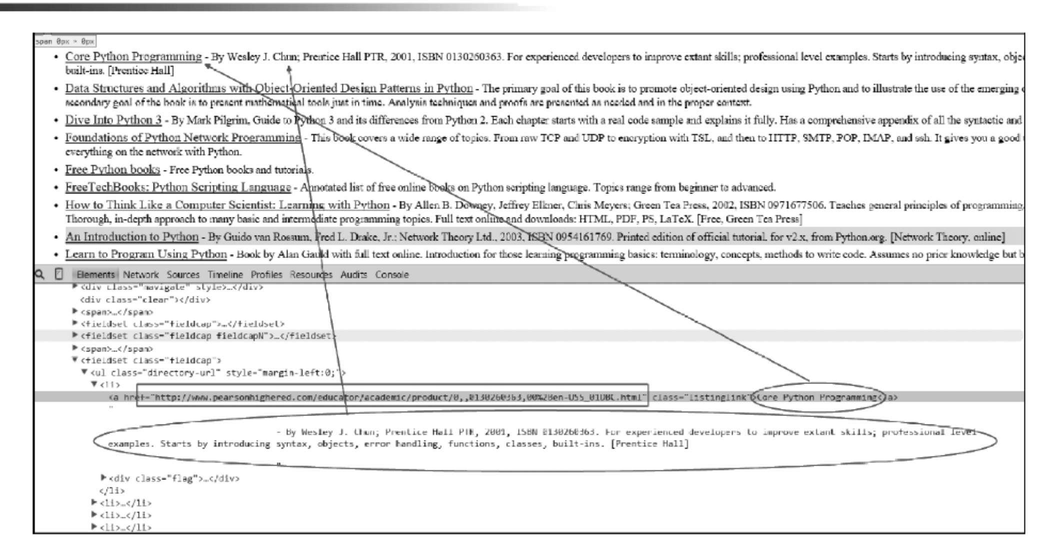



图 14-27　提取数据(一)

```
sel.xpath('//ul/li/a/text()').extract()
```

以及网站的链接：

```
sel.xpath('//ul/li/a/@href').extract()
```

注意，如果这里不加 extract()，xpath()是直接返回一个 Selector 对象组成的列表。写一个循环来打印需要的信息：

```
>>> sites = sel.xpath('//ul/li')
>>> for site in sites:
...     title = site.xpath('a/text()').extract()
...     link = site.xpath('a/@href').extract()
...     desc = site.xpath('text()').extract()
...     print(title, link, desc)
```

实现结果如图 14-28 所示。

```
([u'Top'], [u'/'], [u'\r\n\r\n                              '])
([u'Computers'], [u'/Computers/'], [])
([u'Programming'], [u'/Computers/Programming/'], [])
([u'Languages'], [u'/Computers/Programming/Languages/'], [])
([u'Python'], [u'/Computers/Programming/Languages/Python/'], [])
([], [], [u'\r\n                              ', u'\xa0', u'\r\n
     '])
([u'Computers: Programming: Languages: Python: Resources'], [u'/Computers/Progra
mming/Languages/Python/Resources/'], [u'\r\n                              ', u' \r\n
                 ', u'\r\n              '])
([u'Computers: Programming: Languages: Ruby: Books'], [u'/Computers/Programming/
Languages/Ruby/Books/'], [u'\r\n                              ', u' \r\n
        ', u'\r\n               '])
([u'Deutsch'], [u'/World/Deutsch/Computer/Programmieren/Sprachen/Python/B%C3%BCc
her'], [u'\r\n                  \t', u'\r\n                              ', u'\r\n
              '])
```

图 14-28　提取数据(二)

但是结果不对，它怎么把 Top、Computers、Programming 这些导航栏中的内容也给打印出来啦？查看“审查元素”，原来导航栏也是由<ul><li>标签组成的，如图 14-29 所示。

图 14-29　提取数据(三)

此时,进一步设置 ul 的属性即可:

```
>>> sites = sel.xpath('//ul[@class="directory-url"]/li')
>>> for site in sites:
...     title = site.xpath('a/text()').extract()
...     link = site.xpath('a/@href').extract()
...     desc = site.xpath('text()').extract()
...     print(title, link, desc)
```

实现如图 14-30 所示。

这就是我们想要的代码了,把它放到生产线上实现试试:

```
# 修改 spiders/dmoz_spider.py 文件的 parse()方法
    def parse(self, response):
        # shell 帮我们初始化好 sel,这里我们要自己初始化
        sel = scrapy.selector.Selector(response)
        sites = sel.xpath('//ul[@class="directory-url"]/li')
        for site in sites:
            title = site.xpath('a/text()').extract()
            link = site.xpath('a/@href').extract()
            desc = site.xpath('text()').extract()
            print(title, link, desc)
```

众望所归,如图 14-31 所示。

```
([u'Core Python Programming'], [u'http://www.pearsonhighered.com/educator/academ
ic/product/0,,0130260363,00%2Ben-USS_01DBC.html'], [u'\r\n\t\r\n
                  ', u' \r\n\t\t\t\r\n                              - By Wesley
J. Chun; Prentice Hall PTR, 2001, ISBN 0130260363. For experienced developers to
 improve extant skills; professional level examples. Starts by introducing synta
x, objects, error handling, functions, classes, built-ins. [Prentice Hall]\r\n
                              \r\n                              ', u'\r\n
                           '])
([u'Data Structures and Algorithms with Object-Oriented Design Patterns in Pytho
n'], [u'http://www.brpreiss.com/books/opus7/html/book.html'], [u'\r\n\t\r\n
                              ', u' \r\n\t\t\t\r\n                              -
 The primary goal of this book is to promote object-oriented design using Python
 and to illustrate the use of the emerging object-oriented design patterns.\r\nA
 secondary goal of the book is to present mathematical tools just in time. Analy
sis techniques and proofs are presented as needed and in the proper context.\r\n
                                \r\n                              ', u'\r\n
                             '])
([u'Dive Into Python 3'], [u'http://www.diveintopython.net/'], [u'\r\n\t\r\n
                              ', u' \r\n\t\t\t\r\n
- By Mark Pilgrim, Guide to Python 3  and its differences from Python 2. Each ch
apter starts with a real code sample and explains it fully. Has a comprehensive
appendix of all the syntactic and semantic changes in Python 3\r\n\r\n\r\n
                             \r\n                              ', u'\r\n
                          '])
([u'Foundations of Python Network Programming'], [u'http://rhodesmill.org/brando
n/2011/foundations-of-python-network-programming/'], [u'\r\n\t\r\n
                    ', u' \r\n\t\t\t\r\n                              - This boo
```

图 14-30　提取数据(四)

```
2015-03-01 15:31:28+0800 [dmoz] DEBUG: Crawled (200) <GET http://www.dmoz.org/Co
mputers/Programming/Languages/Python/Resources/> (referer: None)
([u'eff-bot's Daily Python URL'], [u'http://www.pythonware.com/daily/'], [u'\r\n
\t\r\n                                  ', u' \r\n\t\t\t\r\n
                  - Contains links to assorted resources from the Python universe, compi
led by PythonWare.\r\n                                     \r\n
                 ', u'\r\n                                          '])
([u'Free Python and Zope Hosting Directory'], [u'http://www.oinko.net/freepython
/'], [u'\r\n\t\r\n                              ', u' \r\n\t\t\t\r\n
                            - A directory of free Python and Zope hosting providers, w
ith reviews and ratings.\r\n                              \r\n
                ', u'\r\n                              '])
([u"O'Reilly Python Center"], [u'http://oreilly.com/python/'], [u'\r\n\t\r\n
                              ', u' \r\n\t\t\t\r\n
- Features Python books, resources, news and articles.\r\n
          \r\n                              ', u'\r\n
        '])
```

图 14-31　提取数据(五)

## 14.8.10　使用 item

item 其实是自定义的一个容器,用法跟 Python 的字典一样。我们希望 Spiders 将爬取并筛选后的数据存放到 item 容器中,所以 spider 的最终代码应该是这样的:

```
import scrapy

from tutorial.items import DmozItem

class DmozSpider(scrapy.Spider):
    name = "dmoz"
    allowed_domains = ["dmoz.org"]
    start_urls = [
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Books/",
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/"
```

```
    ]

    def parse(self, response):
        sel = scrapy.selector.Selector(response)
        sites = sel.xpath('//ul[@class="directory-url"]/li')
        items = []
        for site in sites:
            item = DmozItem()
            item['title'] = site.xpath('a/text()').extract()
            item['link'] = site.xpath('a/@href').extract()
            item['desc'] = site.xpath('text()').extract()
            items.append(item)
        return items
```

### 14.8.11　存储内容

最简单存储爬取的数据的方式是使用 Feed exports，主要可以导出四种格式：JSON、JSON lines、CSV 和 XML

这里将结果导出为最常用的 JSON 格式：

```
scrapy crawl dmoz -o items.json -t json
```

-o 后面是导出的文件名，-t 指定导出类型。

成功执行命令后，根目录出现了一个叫 items.json 的文件，内容如图 14-32 所示。

至此，我们成功地使用 Scrapy 框架进行了一次完整的爬取操作。


```
1  [{"desc": ["\r\n\t\r\n                                        ", "
   \r\n\t\t\t\r\n                                  - By Wesley J. Chun;
   Prentice Hall PTR, 2001, ISBN 0130260363. For experienced developers to
   improve extant skills; professional level examples. Starts by
   introducing syntax, objects, error handling, functions, classes,
   built-ins. [Prentice Hall]\r\n
   \r\n                                ",
   "\r\n                                "], "link": ["
   http://www.pearsonhighered.com/educator/academic/product/0,,0130260363,00%
   2Ben-USS_01DBC.html"], "title": ["Core Python Programming"]},
2  {"desc": ["\r\n\t\r\n                                       ", "
   \r\n\t\t\t\r\n                                  - The primary goal of this
   book is to promote object-oriented design using Python and to illustrate
   the use of the emerging object-oriented design patterns.\r\nA secondary
   goal of the book is to present mathematical tools just in time. Analysis
   techniques and proofs are presented as needed and in the proper
   context.\r\n
   \r\n                                ",
   "\r\n                                "], "link": ["
   http://www.brpreiss.com/books/opus7/html/book.html"], "title": ["Data
   Structures and Algorithms with Object-Oriented Design Patterns in
   Python"]},
3  {"desc": ["\r\n\t\r\n                                       ", "
   \r\n\t\t\t\r\n                                  - By Mark Pilgrim, Guide
   to Python 3  and its differences from Python 2. Each chapter starts with
   a real code sample and explains it fully. Has a comprehensive appendix
   of all the syntactic and semantic changes in Python
   3\r\n\r\n\r\n
   \r\n                                ",
   "\r\n                                "], "link": ["
   http://www.diveintopython.net/"], "title": ["Dive Into Python 3"]},
4  {"desc": ["\r\n\t\r\n                                       ", "
   \r\n\t\t\t\r\n                                  - This book covers a wide
```


图 14-32　存储内容

# 第15章 GUI的最终选择：Tkinter

到目前为止，几乎所有的 Python 代码都是处于一个文字交互界面的状态。当然，有些崇尚 GEEK 的朋友可能会说："文字就文字呗，Python 本来就应该简单，做一个界面多费事儿啊！"不过，也有另一个帮派在提反对意见："我的用户群体可全都是电脑小白，他们可能会更喜欢友好的界面……"

Python 的 GUI 工具包有很多，之前学习过的 EasyGui 就是其中最简单的一个。不过 EasyGui 实在太简单了，因此它只适合做大家接触 GUI 编程的敲门砖。下面要讲的可不是什么二流的货色了，而是官方御用的 GUI 工具包——Tkinter(IDLE 就是用这个开发的)。

Tkinter 是 Python 的标准 GUI 库，它实际是建立在 Tk 技术上的，如图 15-1 所示。Tk 最初是为 Tcl(这是一门工具命令语言，不是那个电视机品牌)所设计的，但由于其可移植性和灵活性高，加上非常容易使用，因此它逐渐被移植到许多脚本语言中，包括 Perl、Ruby 和 Python。






图 15-1　Tkinter

Tkinter 是 Python 默认的 GUI 库，像 IDLE 就是用 Tkinter 设计出来的，因此直接导入 Tkinter 模块就可以了：

```
>>> import tkinter
```

## 15.1　Tkinter 之初体验

接下来从最简单的例子入手：

```
# p15_1.py
import tkinter as tk

root = tk.Tk()
root.title("FishC Demo")
theLabel = tk.Label(root, text = "我的第二个窗口程序!")
theLabel.pack()
```

```
root.mainloop()
```

执行程序，如图 15-2 所示。

图 15-2 我的第二个窗口程序

代码分析：

```
# 创建一个主窗口，用于容纳整个 GUI 程序
root = tk.Tk()
# 设置主窗口对象的标题栏
root.title("FishC Demo")
# 添加一个 Label 组件，Label 组件是 GUI 程序中最常用的组件之一。
# Label 组件可以显示文本、图标或者图片
# 在这里我们让它显示指定文本
theLabel = tk.Label(root, text = "我的第二个窗口程序!")
# 然后调用 Label 组件的 pack()方法，用于自动调节组件自身的尺寸
theLabel.pack()
# 注意，这时候窗口还是不会显示的…
# 除非执行下面这条代码!
root.mainloop()
```

tkinter.mainloop()通常是你程序的最后一行代码，执行后程序进入主事件循环。学习过界面编程的朋友应该有听过一句名言“Don't call me, I will call you.”，意思是一旦进入了主事件循环，就由 Tkinter 掌管一切了。现在不理解没关系，在后面的学习中你会有深刻的体会。GUI 程序的开发与以往的开发经验会有截然不同的感受。

## 进阶版本

通常如果要写一个比较大的程序，那么应该先把代码给封装起来。在面向对象的编程语言中，就是封装成类。看下面进阶版的例子：

```
# p15 - 2.py
import tkinter as tk

class App:
    def __init__(self, root):
        frame = tk.Frame(root)
        frame.pack()
        self.hi_there = tk.Button(frame, text = "打招呼", fg = command = self.say_hi)
        self.hi_there.pack(side = tk.LEFT)

    def say_hi(self):
        print("互联网的广大朋友们大家好,我是小甲鱼!")

root = tk.Tk()
app = App(root)

root.mainloop()
```

程序跑起来后出现一个“打招呼”按钮，单击它就能从 IDLE 接收到回馈信息，如图 15-3 所示。

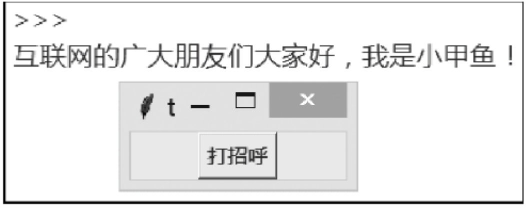



图 15-3 进阶版本(一)

代码分析：

```
import tkinter as tk

class App:
    def __init__(self, root):
        # 创建一个框架,然后在里边添加一个 Button 按钮组件
        # 框架一般是用于在复杂的布局中起到将组件分组的作用
        frame = tk.Frame(root)
        frame.pack()
        # 创建一个按钮组件,fg 是 foreground 的缩写,就是设置前景色的意思
        self.hi_there = tk.Button(frame, text = "打招呼", fg = "blue", command = self.say_hi)
        self.hi_there.pack()

    def say_hi(self):
        print("互联网的广大朋友们大家好,我是小甲鱼!")

# 创建一个 toplevel 的根窗口,并把它作为参数实例化 app 对象
root = tk.Tk()
app = App(root)
# 开始主事件循环
root.mainloop()
```

可以通过修改 pack()方法的 side 参数，side 参数可以设置 LEFT、RIGHT、TOP 和 TOTTOM 四个方位，默认的设置是 side=tkinter.TOP。

例如可以修改为左对齐 frame.pack(side=tk.LEFT)，修改后程序如图 15-4 所示。

如果你不想按钮挨着“墙角”，可以通过设置 pack()方法的 padx 和 pady 参数自定义按钮的偏移位置：

```
frame.pack(side = tk.LEFT, padx = 10, pady = 10)
```

修改后程序如图 15-5 所示。

按钮既然可以设置前景色，那一定也能设置背景色吧？没错，bg 参数就是 background 背景色的缩写：

```
self.hi_there = tk.Button(frame, text = "打招呼", bg = "black", fg = "white", command = self.say_hi)
```

修改后程序如图 15-6 所示。




图 15-4　进阶版本(二)

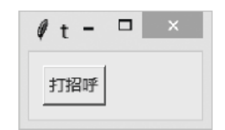



图 15-5　进阶版本(三)




图 15-6　进阶版本(四)

## 15.2　Label 组件



Label 组件是用于在界面上输出描述的标签，例如提示用户“您所下载的影片含有未成年人限制内容，请满 18 岁后再点击观看！”，如图 15-7 所示。



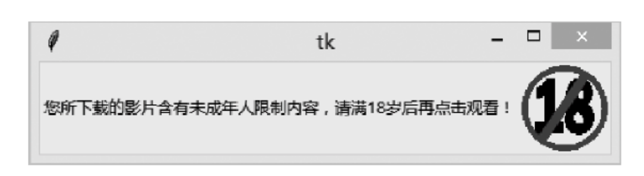



图 15-7　Label 组件(一)

```
# p15_3.py
from tkinter import *
# 导入 tkinter 模块的所有内容

root = Tk()
# 创建一个文本 Label 对象
textLabel = Label(root, \
text = "您所下载的影片含有未成年人限制内容,请满 18 岁后再点击观看!")
textLabel.pack(side = LEFT)
# 创建一个图像 Label 对象
# 用 PhotoImage 实例化一个图片对象(支持 gif 格式的图片)
photo = PhotoImage(file = "18.gif")
imgLabel = Label(root, image = photo)
imgLabel.pack(side = RIGHT)

mainloop()
```

可以直接在字符串中使用\n 对现实中的文本进行断行，如图 15-8 所示。

如果想将文字部分左对齐，并在水平位置与边框留有一定的距离，只需要设置 Label 的 justify 和 padx 选项即可：

```
textLabel = Label(root,
text = "您所下载的影片含有未成年人限制内容,\n 请满 18 岁后再点击观看!",
justify = LEFT, padx = 10)
```

程序实现如图 15-9 所示。

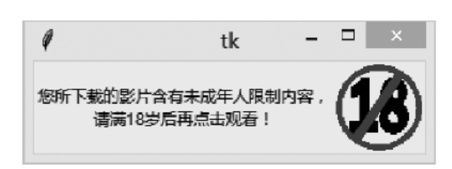



图 15-8　Label 组件(二)

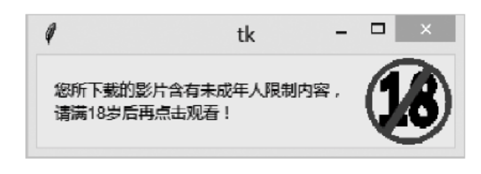



图 15-9　Label 组件(三)

有时候可能需要将图片和文字分开，例如将图片作为背景，文字显示在图片的上面，只需要设置 compound 选项即可：

```
# p15_4.py
from tkinter import *

root = Tk()
photo = PhotoImage(file = "bg.gif")
theLabel = Label(root,
                 text = "学 Python\n 到 FishC",
```

```
                          justify = LEFT,
                          image = photo,
                          compound = CENTER, # 设置文本和图像的混合模式
                          font = ("华康少女字体", 20), # 设置字体和字号
                          fg = "white" # 设置文本颜色
                          )
theLabel.pack()

mainloop()
```

程序实现如图 15-10 所示。

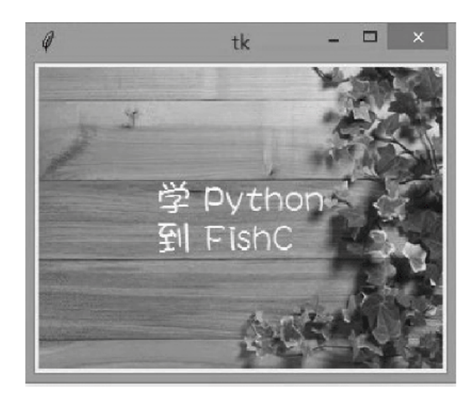



图 15-10　Label 组件(四)

## 15.3　Button 组件

Button 组件是用于实现一个按钮,它的绝大多数选项跟 Label 组件是一样的。不过 Button 组有一个 Label 组件实现不了的功能,那就是可以接收用户的信息。Button 组件有一个 command 选项,用于指定一个函数或方法,当用户单击按钮的时候,Tkinter 就会自动地去调用这个函数或方法了。

下面修改第一个例子,添加一个按钮,在按钮被单击之后 Label 文本发生改变。想要文本发生改变,只需要设置 textvariable 选项为 Tkinter 变量即可:

```
# p15_5.py
from tkinter import *

def callback():
    var.set("吹吧你,我才不信呢~")

root = Tk()
frame1 = Frame(root)
frame2 = Frame(root)
# 创建一个文本 Label 对象
var = StringVar()
var.set("您所下载的影片含有未成年人限制内容,\n 请满 18 岁后再点击观看!")
textLabel = Label(frame1,
                  textvariable = var,
```

```
                    justify = LEFT)
textLabel.pack(side = LEFT)
# 创建一个图像 Label 对象
# 用 PhotoImage 实例化一个图片对象(支持 gif 格式的图片)
photo = PhotoImage(file = "18.gif")
imgLabel = Label(frame1, image = photo)
imgLabel.pack(side = RIGHT)
# 加一个按钮
theButton = Button(frame2, text = "已满 18 周岁", command = callback)
theButton.pack()
frame1.pack(padx = 10, pady = 10)
frame2.pack(padx = 10, pady = 10)

mainloop()
```

## 15.4 Checkbutton 组件



Checkbutton 组件就是常见的多选按钮，而 Radiobutton 则是单选按钮。

```
# p15_6.py
from tkinter import *

root = Tk()
# 需要一个 Tkinter 变量,用于表示该按钮是否被选中
v = IntVar()
c = Checkbutton(root, text = "测试一下", variable = v)
c.pack()
# 如果选项被选中,那么变量 v 被赋值为 1,否则为 0
# 可以用个 Label 标签动态地给大家展示:
l = Label(root, textvariable = v)
l.pack()

mainloop()
```

程序实现如图 15-11 所示。

当单击选项时，Label 显示的变量相应地发生了改变，如图 15-12 所示。

有了前面的基础，下面写一个古代四大美女的程序：

```
from tkinter import *

root = Tk()
GIRLS = ["西施", "王昭君", "貂蝉", "杨玉环"]
v = []
for girl in GIRLS:
    v.append(IntVar())
    b = Checkbutton(root, text = girl, variable = v[-1])
    b.pack()

mainloop()
```

程序实现如图 15-13 所示。




图 15-11　Checkbutton 组件(一)

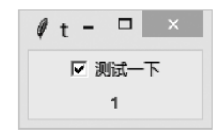



图 15-12　Checkbutton 组件(二)




图 15-13　古代四大美女的程序

这里应该把所有的 Checkbutton 组件都向左对齐一下会比较好看,通过设置 pack()方法的 anchor 选项可以实现。anchor 选项是用于指定显示位置,可以设置为 N、NE、E、SE、S、SW、W、NW 和 CENTER 九个不同的值。相信地理学得不错的朋友一下子就反应过来了,它们正是东西南北的缩写,然后按照地图上的"上北下南左西右东"的原则,这样就可以定位要显示的位置了,如图 15-14 所示。

这里要"左对齐",也就是设置 b.pack(anchor=W),修改后如图 15-15 所示。




图 15-14　anchor 选项

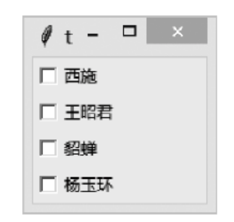



图 15-15　修改后的程序

## 15.5　Radiobutton 组件

Radiobutton 组件跟 Checkbutton 组件的用法基本一致,唯一不同的是 Radiobutton 实现的是"单选"的效果。要实现这种互斥的效果,同一组内的所有 Radiobutton 只能共享一个 variable 选项,并且需要设置不同的 value 选项值:

```
# p15_8.py
from tkinter import *

root = Tk()
```

```
v = IntVar()
Radiobutton(root, text = "One", variable = v, value = 1).pack(anchor = W)
Radiobutton(root, text = "Two", variable = v, value = 2).pack(anchor = W)
Radiobutton(root, text = "Three", variable = v, value = 3).pack(anchor = W)

mainloop()
```

程序实现如图 15-16 所示。

如果有多个选项，可以使用循环来处理，这会使得代码更加简洁：

```
# p15_9.py
from tkinter import *

root = Tk()
LANGS = [
    ("Python", 1),
    ("Perl", 2),
    ("Ruby", 3),
    ("Lua", 4)]
v = IntVar()
v.set(1)
for lang, num in LANGS:
    b = Radiobutton(root, text = lang, variable = v, value = num)
    b.pack(anchor = W)

mainloop()
```

程序实现如图 15-17 所示。

在此，如果你不喜欢前面这个小圈圈，还可以改成按钮的形式：

```
# 将 indicatoron 设置为 False 即可去掉前面的小圆圈
b = Radiobutton(root, text = lang, variable = v, value = num, indicatoron = False)
b.pack(fill = X)
```

程序修改后如图 15-18 所示。

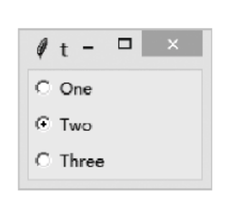



图 15-16 Radiobutton 组件(一)

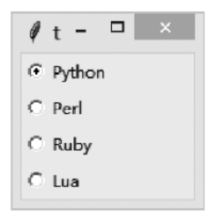



图 15-17 Radiobutton 组件(二)

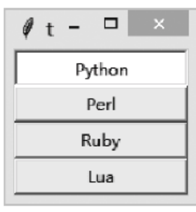



图 15-18 Radiobutton 组件(三)

## 15.6 LabelFrame 组件

LabelFrame 组件是 Frame 框架的进化版，从形态上来说，也就是添加了 Label 的 Frame，但有了它，Checkbutton 和 Radiobutton 的组件分组就变得简单了：

```
# p15_10.py
from tkinter import *

root = Tk()
group = LabelFrame(root, text = "最好的脚本语言是?", padx = 5, pady = 5)
group.pack(padx = 10, pady = 10)
LANGS = [
    ("Python", 1),
    ("Perl", 2),
    ("Ruby", 3),
    ("Lua", 4)]
v = IntVar()
v.set(1)
for lang, num in LANGS:
    b = Radiobutton(group, text = lang, variable = v, value = num)
    b.pack(anchor = W)

mainloop()
```

程序实现如图 15-19 所示。

图 15-19　LabelFrame 组件

## 15.7　Entry 组件



Entry 组件就是平时所说的输入框。输入框是跟程序打交道的一个途径,例如程序要求你输入账号密码,那么它就需要提供两个输入框给你,用于接收密码的输入框还会用星号将实际输入的内容隐藏起来。

学了前面好几个 Tkinter 的组件之后应该不难发现——其实很多方法和选项它们之间都是通用的。例如在输入框中用代码添加和删除内容,同样也是使用 insert()和 delete()方法:

```
# p15_11.py
from tkinter import *

root = Tk()
e = Entry(root)
e.pack(padx = 20, pady = 20)
e.delete(0, END)
e.insert(0, "默认文本…")

mainloop()
```

程序实现如图 15-20 所示。

获取输入框里边的内容,可以使用 Entry 组件的 get()方法。当然也可以将一个 Tkinter 的变量(通常是 StringVar)挂钩到 textvariable 选项,然后通过变量的 get()方法获取。




图 15-20　Entry 组件(一)

在下面的例子中,添加一个按钮,当单击按钮的时候,获取输入框的内容并打印出来,然后清空输入框。

程序实现起来如图 15-21 所示。

单击"获取信息"按钮,在 IDLE 中将输入框中的内容显示出来,如图 15-22 所示。



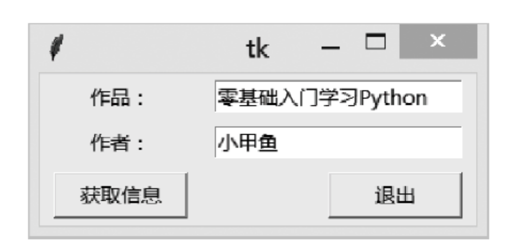


图 15-21 Entry 组件(二)



图 15-22 Entry 输入框

```
# p15_12.py
from tkinter import *

root = Tk()
# Tkinter 总共提供了三种布局组件的方法: pack(),gird()和 place()
# grid()方法允许你用表格的形式来管理组件的位置
# row 选项代表行,column 选项代表列
# 例如 row = 1,column = 2 表示第二行第三列(0 表示第一行)
Label(root, text = "作品: ").grid(row = 0)
Label(root, text = "作者: ").grid(row = 1)
e1 = Entry(root)
e2 = Entry(root)
e1.grid(row = 0, column = 1, padx = 10, pady = 5)
e2.grid(row = 1, column = 1, padx = 10, pady = 5)

def show():
    print("作品: «%s»" % e1.get())
    print("作者: %s" % e2.get())
    e1.delete(0, END)
    e2.delete(0, END)

# 如果表格大于组件,那么可以使用 sticky 选项来设置组件的位置
# 同样你需要使用 N,E,S,W 以及它们的组合 NE,SE,SW,NW 来表示方位
Button(root, text = "获取信息", width = 10, command = show)\
             .grid(row = 3, column = 0, sticky = W, padx = 10, pady = 5)
Button(root, text = "退出", width = 10, command = root.quit)\
             .grid(row = 3, column = 1, sticky = E, padx = 10, pady = 5)

mainloop()
```

你可能会遇到问题：为什么单击“退出”按钮没有反应？这是因为之前也提到过，Python 的 IDLE 事实上也是使用 Tkinter 设计的，因此当程序是使用 IDLE 运行的时候，就会出现此类冲突。解决的方法也很简单，只需要直接双击打开程序即可。

如果想设计一个密码输入框，即使用星号(*)代替用户输入的内容，只需要设置 show 选项即可：

```
# p15_13.py
from tkinter import *

root = Tk()
Label(root, text = "账号: ").grid(row = 0)
```

```
Label(root, text = "密码: ").grid(row = 1)
v1 = StringVar()
v2 = StringVar()
e1 = Entry(root, textvariable = v1)
e2 = Entry(root, textvariable = v2, show = " * ")
e1.grid(row = 0, column = 1, padx = 10, pady = 5)
e2.grid(row = 1, column = 1, padx = 10, pady = 5)

def show():
    print("账号: %s" % v1.get())
    print("密码: %s" % v2.get())
    e1.delete(0, END)
    e2.delete(0, END)

Button(root, text = "芝麻开门", width = 10, command = show)\
              .grid(row = 3, column = 0, sticky = W, padx = 10, pady = 5)
Button(root, text = "退出", width = 10, command = root.quit)\
              .grid(row = 3, column = 1, sticky = E, padx = 10, pady = 5)

mainloop()
```

程序实现如图 15-23 所示。

单击“芝麻开门”可以得到密码的信息，如图 15-24 所示。

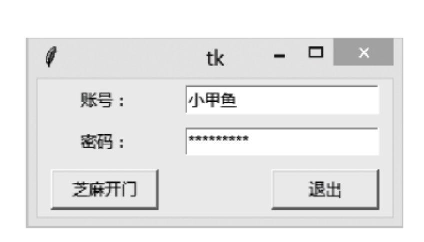



图 15-23　Entry 组件(三)




图 15-24　Entry 组件(四)

另外，Entry 组件还支持验证输入内容的合法性。例如输入框要求输入的是数字，用户输入了字母那就属于“非法”。实现该功能，需要通过设置 validate、validatecommand 和 invalidcommand 三个选项。

首先启用验证的“开关”是 validate 选项，该选项可以设置的值如表 15-1 所示。

**表 15-1　validate 选项可以设置的值**

| 值 | 含　义 |
|---|---|
| 'focus' | 当 Entry 组件获得或失去焦点的时候验证 |
| 'focusin' | 当 Entry 组件获得焦点的时候验证 |
| 'focusout' | 当 Entry 组件失去焦点的时候验证 |
| 'key' | 当输入框被编辑的时候验证 |
| 'all' | 当出现上面任何一种情况的时候验证 |
| 'none' | 关闭验证功能。默认设置该选项(即不启用验证)。注意，是字符串的'none'，而非 None |

其次是为 validatecommand 选项指定一个验证函数，该函数只能返回 True 或 False 表示验证的结果。一般情况下验证函数只需要知道输入框的内容即可，可以通过 Entry 组件的 get() 方法获得该字符串。

在下面的例子中，在第一个输入框中输入“小甲鱼”并通过 Tab 键将焦点转移到第二个输入框的时候，验证功能被成功触发：

```
# p15_14.py
from tkinter import *

root = Tk()

def test():
    if e1.get() == "小甲鱼":
        print("正确!")
        return True
    else:
        print("错误!")
        e1.delete(0, END)
        return False

v = StringVar()
e1 = Entry(root, textvariable=v, validate="focusout", validatecommand=test)
e2 = Entry(root)
e1.pack(padx=10, pady=10)
e2.pack(padx=10, pady=10)

mainloop()
```

程序实现如图 15-25 所示。

最后，invalidcommand 选项指定的函数只有在 validatecommand 的返回值为 False 的时候才被调用。

在下面的例子中，在第一个输入框中输入“小鱿鱼”，并通过 Tab 键将焦点转移到第二个输入框，validatecommand 指定的验证函数被触发并返回 False，接着 invalidcommand 被触发：

```
# p15_15.py
…
def test2():
    print("我被调用了……")
    return True

e1 = Entry(master, textvariable=v, validate="focusout", validatecommand=test1, invalidcommand=test2)
…
```

程序修改后实现如图 15-26 所示。




图 15-25 Entry 组件（五）




图 15-26 Entry 组件（六）

其实，Tkinter 还有个“隐藏技能”——Tkinter 为验证函数提供一些额外的选项，如表 15-2 所示。

**表 15-2　Tkinter 为验证函数提供一些额外的选项**

| 选项 | 含　义 |
| --- | --- |
| '%d' | 操作代码：0 表示删除操作；1 表示插入操作；2 表示获得、失去焦点或 textvariable 变量的值被修改 |
| '%i' | 当用户尝试插入或删除操作的时候，该选项表示插入或删除的位置(索引号)<br>如果是由于获得、失去焦点或 textvariable 变量的值被修改而调用验证函数，那么该值是－1 |
| '%P' | 当输入框的值允许改变的时候，该值有效<br>该值为输入框的最新文本内容 |
| '%s' | 该值为调用验证函数前输入框的文本内容 |
| '%S' | 当插入或删除操作触发验证函数的时候，该值有效<br>该选项表示文本被插入和删除的内容 |
| '%v' | 该组件当前的 validate 选项的值 |
| '%V' | 调用验证函数的原因<br>该值是 'focusin'，'focusout'，'key'或 'forced'(textvariable 选项指定的变量值被修改)中的一个 |
| '%W' | 该组件的名字 |

为了使用这些选项，你可以这样写：

```
validatecommand = (f, s1, s2, ...)
```

其中，f 是验证函数名，s1、s2、s3 是额外的选项，这些选项会作为参数依次传给 f 函数。在此之前，需要调用 register()方法将验证函数包装起来：

```
# p15_16.py
from tkinter import *

root = Tk()
v = StringVar()

def test(content, reason, name):
    if content == "小甲鱼":
        print("正确!")
        print(content, reason, name)
        return True
    else:
        print("错误!")
        print(content, reason, name)
        return False

testCMD = root.register(test)
e1 = Entry(root, textvariable = v, validate = "focusout", validatecommand = (testCMD, '%P', '%v', '%W'))
e2 = Entry(root)
e1.pack(padx = 10, pady = 10)
e2.pack(padx = 10, pady = 10)

mainloop()
```

程序实现如图 15-27 所示。

下面实现一个简单的计算器：

```
# p15_17.py
from tkinter import *
```

```
root = Tk()
frame = Frame(root)
frame.pack(padx=10, pady=10)
v1 = StringVar()
v2 = StringVar()
v3 = StringVar()

def test(content):
    # 注意，这里你不能使用 e1.get()或者 v1.get()来获取输入的内容
    # 因为 validate 选项指定为"key"的时候，有任何输入操作都会被拦截到这个函数中
    # 也就是说先拦截，只有这个函数返回 True，那么输入的内容才会到变量里边
    # 所以要使用 %P 来获取最新的输入框内容
    if content.isdigit():
        return True
    else:
        return False

testCMD = root.register(test)
Entry(frame, textvariable=v1, width=10, validate="key", \
           validatecommand=(testCMD, '%P')).grid(row=0, column=0)
Label(frame, text="+").grid(row=0, column=1)
Entry(frame, textvariable=v2, width=10, validate="key", \
           validatecommand=(testCMD, '%P')).grid(row=0, column=2)
Label(frame, text="=").grid(row=0, column=3)
Entry(frame, textvariable=v3, width=10, validate="key", \
           validatecommand=(testCMD, '%P')).grid(row=0, column=4)

def calc():
    result = int(v1.get()) + int(v2.get())
    v3.set(result)

Button(frame, text="计算结果", command=calc).grid(row=1, column=2, pady=5)

mainloop()
```

程序实现如图 15-28 所示。

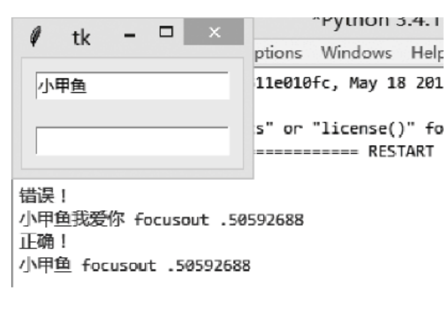



图 15-27　Entry 组件（七）




图 15-28　Entry 组件（八）

## 15.8　Listbox 组件



如果需要提供选项给用户选择，单选可以用 Radiobutton 组件，多选则可以用 Checkbutton 组件。但如果提供的选项非常多，例如选择你所在的城市，通过 Radiobutton 和

Checkbutton 组件来实现直接导致的结果就是：用户界面不够存放那么多按钮！

这时候就可以考虑使用 Listbox 组件，Listbox 是以列表的形式显示出来，并支持滚动条操作，所以对于在需要提供大量选项的情况下会更适用一些。

当创建一个 Listbox 组件的时候，它是空的（里边什么都没有）。所以，首先要做的第一件事就是添加一行或多行文本进去。使用 insert()方法添加文本，该方法有两个参数：第一个参数是插入的索引号，第二个参数是插入的字符串。索引号通常是项目的序号（第一项的序号是 0）。

当然对于多个项目，应该使用循环：

```
#p15-18.py
from tkinter import *

root = Tk()
# 创建一个空列表
theLB = Listbox(root, setgrid=True)
theLB.pack()
# 往列表里添加数据
for item in ["鸡蛋", "鸭蛋", "鹅蛋", "李狗蛋"]:
    theLB.insert(END, item)
theButton = Button(root, text="删除", command=lambda x=theLB: x.delete(ACTIVE))
theButton.pack()

mainloop()
```

程序实现如图 15-29 所示。



图 15-29　Listbox 组件(一)

使用 delete()方法删除列表中的项目，最常用的操作是删除列表中的所有项目：listbox.delete(0, END)

当然也可以删除指定的项目，下边添加一个独立按钮来删除 ACTIVE 状态的项目：

```
# 跟 END 一样,这个 ACTIVE 是一个特殊的索引号,表示当前被选中的项目)
theButton = Button(master, text = "删除", command = lambda x =
theLB: x.delete(ACTIVE))
theButton.pack()
```

最后，这个 Listbox 组件根据 selectmode 选项提供了四种不同的选择模式：SINGLE（单选）、BROWSE（也是单选，但拖动鼠标或通过方向键可以直接改变选项）、MULTIPLE（多选）和 EXTENDED（也是多选，但需要同时按住 Shift 键或 Ctrl 键或拖动光标实现）。默认的选择模式是 BROWSE。

选项增多麻烦事儿就接踵而来，例如你发现 Listbox 组件默认只能显示 10 个项目，而你手头有 11 个项目：

```
# p15_19.py
from tkinter import *

root = Tk()
# 创建一个空列表
theLB = Listbox(root, setgrid=True)
theLB.pack()
# 往列表里添加数据
for item in range(11):
    theLB.insert(END, item)
```

```
mainloop()
```

程序实现如图 15-30 所示。

虽然说利用鼠标滚轮可以迫使最后一个项目“现身”，但这样往往很容易被用户忽略……

有两个方法可以解决上述问题，第一方法就是修改 height 选项：

```
theLB = Listbox(master, height = 11)
```

修改后程序如图 15-31 所示。

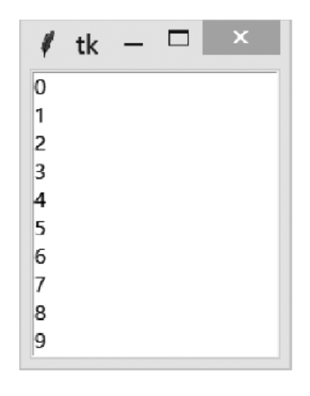



图 15-30 Listbox 组件(二)

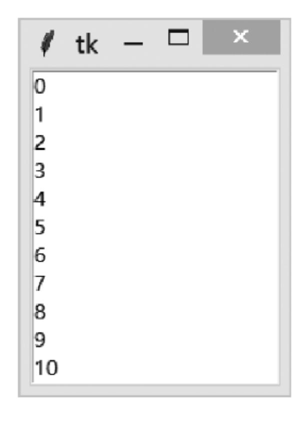



图 15-31 Listbox 组件(三)

修改 height 选项固然可以达到我们的目的，但如果项目太多(例如一百多个)，这个方法就不适用了(导致列表框太长)。还有一个方法更灵活，就是为 Listbox 组件添加滚动条。

## 15.9 Scrollbar 组件

虽然滚动条是作为一个独立的组件存在，不过平时它都是几乎与其他组件配合使用的。下面例子演示如何使用垂直滚动条。

为了在某个组件上安装垂直滚动条，需要做两件事：

(1) 设置该组件的 yscrollbarcommand 选项为 Scrollbar 组件的 set()方法；

(2) 设置 Scrollbar 组件的 command 选项为该组件的 yview()方法。

```
# p15_20.py
from tkinter import *

root = Tk()
sb = Scrollbar(root)
sb.pack(side = RIGHT, fill = Y)
lb = Listbox(root, yscrollcommand = sb.set)
for i in range(1000):
    lb.insert(END, str(i))
lb.pack(side = LEFT, fill = BOTH)
sb.config(command = lb.yview)

mainloop()
```

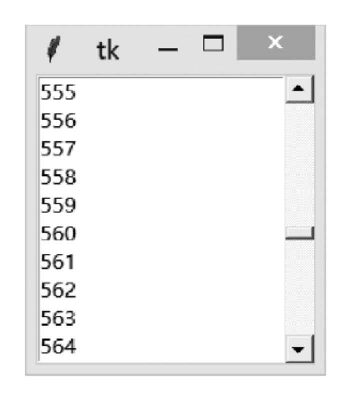
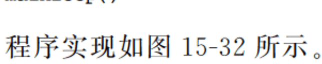



图 15-32 Scrollbar 组件



程序实现如图 15-32 所示。

分析：事实上这是一个互联互通的过程。当用户操作滚动条进行滚动的时候，滚动条响应滚动并同时通过 Listbox 组件的 yview()方法滚动列表框里的内容；同样，当列表框中可视范围发生改变的时候，Listbox 组件通过调用 Scrollbar 组件的 set()方法设置滚动条的最新位置。

## 15.10 Scale 组件

Scale 组件跟 Scrollbar 滚动条组件很相似——都可以滚、都有滑块、都是条形……但它们的使用范围可不尽相同。Scale 组件主要是通过滑块来表示某个范围内的一个数字，可以通过修改选项设置范围以及分辨率(精度)。

当希望用户输入某个范围内的一个数值，使用 Scale 组件可以很好地代替 Entry 组件。创建一个指定范围的 Scale 组件其实非常容易，只需要指定它的 from 和 to 两个选项即可。但由于 from 本身是 Python 的关键字，所以为了区分需要在后边紧跟一个下划线，如 from_。

```
# p15_21.py
from tkinter import *

root = Tk()
Scale(root, from_ = 0, to = 42).pack()
Scale(root, from_ = 0, to = 200, orient = HORIZONTAL).pack()

mainloop()
```

程序实现如图 15-33 所示。

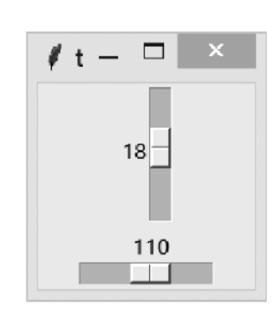


图 15-33　Scale 组件(一)

使用 get()方法可以获取当前滑块的位置：

```
# p15_22.py
from tkinter import *

root = Tk()
s1 = Scale(root, from_ = 0, to = 42)
s1.pack()
s2 = Scale(root, from_ = 0, to = 200, orient = HORIZONTAL)
s2.pack()

def show():
    print(s1.get(), s2.get())

Button(root, text = "获得位置", command = show).pack()

mainloop()
```

程序实现如图 15-34 所示。

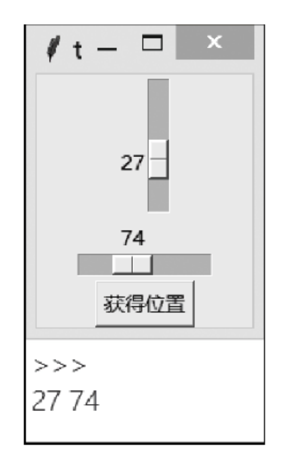
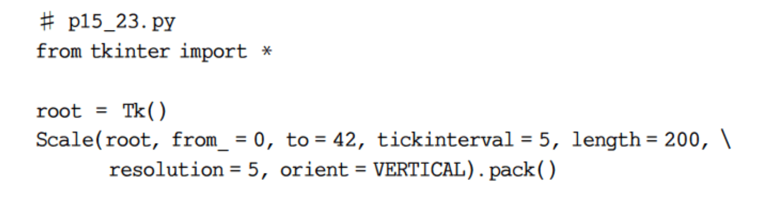


图 15-34　Scale 组件(二)

可以通过 resolution 选项控制分辨率(步长)，通过 tickinterval 选项设置刻度：



```
# p15_23.py
from tkinter import *

root = Tk()
Scale(root, from_ = 0, to = 42, tickinterval = 5, length = 200, \
      resolution = 5, orient = VERTICAL).pack()
```



```
Scale(root, from_ = 0, to = 200, tickinterval = 10, length = 600, \
      orient = HORIZONTAL).pack()

mainloop()
```

程序实现如图 15-35 所示。

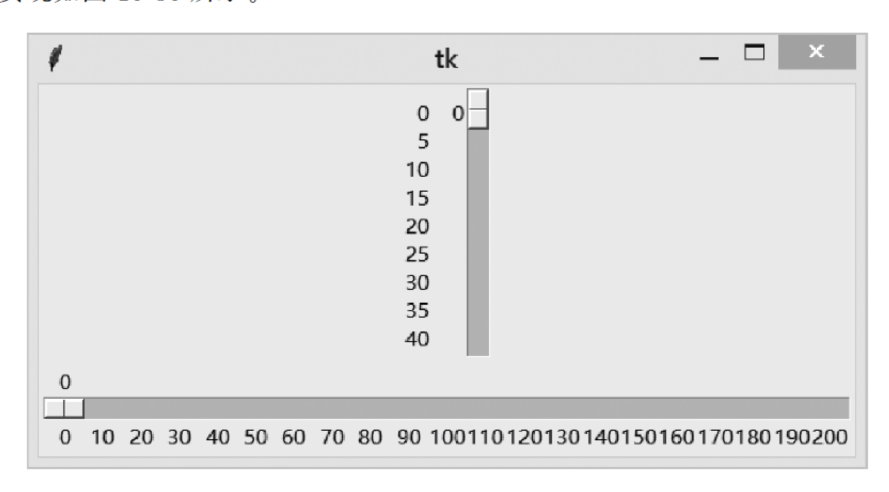



图 15-35 Scale 组件(三)

## 15.11 Text 组件



截至目前，我们已经学了不少组件：绘制单行文本使用 Label 组件，多行选项使用 Listbox 组件，输入框使用 Entry 组件，按钮使用 Button 组件，还有 Radiobutton 和 Checkbutton 组件用于提供单选或多选的情况。多个组件可以用 Frame 组件先搭建一个框架，这样组织起来会更加有条不紊。最后还学习了两个会滚动的组件：Scrollbar 和 Scale。Scrollbar 组件用于实现滚动条，而 Scale 则是让用户在一个范围内选择一个确定的值。

Text(文本)组件用于显示和处理多行文本。在 Tkinter 的所有组件中，Text 组件显得异常强大和灵活，它适用于处理多种任务。虽然该组件的主要目的是显示多行文本，但它常常也被用于作为简单的文本编辑器和网页浏览器使用。

当创建一个 Text 组件的时候，它里面是没有内容的。为了给其插入内容，可以使用 insert()方法以及 INSERT 或 END 索引号：

```
# p15_24.py
from tkinter import *

root = Tk()
text = Text(root, width = 30, height = 2)
text.pack()
# INSERT 索引表示插入光标当前的位置
text.insert(INSERT, "I love\n")
text.insert(END, "FishC.com!")

mainloop()
```

程序实现如图 15-36 所示。

Text 组件不仅支持插入和编辑文本，它还支持插入 image 对象和 window 组件：

```
# p15_25.py
from tkinter import *

root = Tk()
text = Text(root, width = 20, height = 5)
text.pack()
text.insert(INSERT, "I love FishC.com!")

def show():
    print("哟,我被点了一下～")

b1 = Button(text, text = "点我点我", command = show)
text.window_create(INSERT, window = b1)

mainloop()
```

程序实现如图 15-37 所示。

下面的代码将实现单击一下按钮显示一张图片的功能：

```
# p15_26.py
from tkinter import *

root = Tk()
text = Text(root, width = 30, height = 10)
text.pack()
text.insert(INSERT, "I love FishC.com!")
photo = PhotoImage(file = 'fishc.gif')

def show():
    text.image_create(END, image = photo)

b1 = Button(text, text = "点我点我", command = show)
text.window_create(INSERT, window = b1)

mainloop()
```

程序实现如图 15-38 所示。

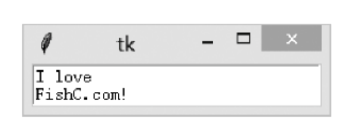



图 15-36　Text 组件(一)




图 15-37　Text 组件(二)




图 15-38　Text 组件(三)

## 15.11.1 Indexes 用法

Indexes(索引)是用来指向 Text 组件中文本的位置，跟 Python 的序列索引一样，Text 组件索引也是对应实际字符之间的位置。

Tkinter 提供一系列不同的索引类型：

- "line. column"(行/列)。
- "line. end"(某一行的末尾)。
- INSERT。
- CURRENT。
- END。
- user-defined marks。
- user-defined tags("tag. first","tag. last")。
- selection(SEL_FIRST,SEL_LAST)。
- window coordinate("@x,y")。
- embedded object name(window,images)。
- expressions。

### 1. "line. column"

用行号和列号组成的字符串是常用的索引方式，它们将索引位置的行号和列号以字符串的形式表示出来(中间以"."分隔，例如"1.0")。需要注意的是，行号以 1 开始，列号则以 0 开始。还可以使用以下语法构建索引：

```
"%d.%d" % (line, column)
```

指定超出现有文本的最后一行的行号，或超出一行中列数的列号都不会引发错误。对于这样的指定，Tkinter 解释为已有内容的末尾的下一个位置。

需要注意的是，使用“行/列”的索引方式看起来像是浮点值。其实在需要指定索引的时候使用浮点值代替也是可以的：

```
text.insert(INSERT, "I love FishC")
print(text.get("1.2", 1.6))
```

程序实现如图 15-39 所示。

### 2. "line. end"

行号加上字符串". end"的格式表示为该行最后一个字符的位置：

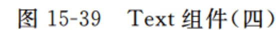
```
text.insert(INSERT, "I love FishC")
print(text.get("1.2", "1.end"))
```

程序实现如图 15-40 所示。

### 3. INSERT(或"insert")

对应插入光标的位置。





图 15-39　Text 组件(四)

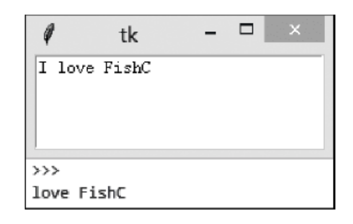



图 15-40　Text 组件(五)

#### 4. CURRENT(或"current")

对应与鼠标坐标最接近的位置。不过,如果你紧按鼠标任何一个按钮,会直到你松开它才响应。

#### 5. END(或"end")

对应 Text 组件的文本缓冲区最后一个字符的下一个位置。

#### 6. user-defined marks

user-defined marks 是对 Text 组件中位置的命名。INSERT 和 CURRENT 是两个预先命名好的 marks,除此之外可以自定义 marks。

#### 7. User-defined tags

User-defined tags 代表可以分配给 Text 组件的特殊事件绑定和风格。

可以使用"tag.first"(使用 tag 的文本的第一个字符之前)和"tag.last"(使用 tag 的文本的最后一个字符之后)语法表示标签的范围:

```
"%s.first" % tagname
"%s.last" % tagname
```

#### 8. selection(SEL_FIRST,SEL_LAST)

selection 是一个名为 SEL(或"sel")的特殊 tag,表示当前被选中的范围,可以使用 SEL_FIRST 到 SEL_LAST 来表示这个范围。如果没有选中的内容,那么 Tkinter 会抛出一个 TclError 异常。

#### 9. window coordinate("@x,y")

可以使用窗口坐标作为索引。例如在一个事件绑定中,你可以使用以下代码找到最接近鼠标位置的字符:

```
"@%d,%d" % (event.x, event.y)
```

#### 10. embedded object name (window,images)

embedded object name 用于指向在 Text 组件中嵌入的 window 和 image 对象。要引用

一个 window，只要简单地将一个 Tkinter 组件实例作为索引即可。引用一个嵌入的 image，只需使用相应的 PhotoImage 和 BitmapImage 对象。

### 11. expressions

expressions 用于修改任何格式的索引，用字符串的形式实现修改索引的表达式。具体表达式实现如表 15-3 所示。

**表 15-3 expressions**

| 表达式 | 含 义 |
|---|---|
| "+ count chars" | 将索引向前(->)移动 count 个字符。可以越过换行符，但不能超过 END 的位置 |
| "- count chars" | 将索引向后(<-)移动 count 个字符。可以越过换行符，但不能超过"1.0"的位置 |
| "+ count lines" | 将索引向前(->)移动 count 行。索引会尽量保持与移动前在同一列上，但如果移动后的那一行字符太少，将移动到该行的末尾 |
| "- count lines" | 将索引向后(<-)移动 count 行。索引会尽量保持与移动前在同一列上，但如果移动后的那一行字符太少，将移动到该行的末尾 |
| " linestart" | 将索引移动到当前索引所在行的起始位置。注意：使用该表达式前边必须用一个空格隔开 |
| " lineend" | 将索引移动到当前索引所在行的末尾。注意：使用该表达式前边必须用一个空格隔开 |
| " wordstart" | 将索引移动到当前索引指向的单词的开头。单词的定义是一系列字母、数字、下划线或任何非空白字符的组合。注意：使用该表达式前边必须用一个空格隔开 |
| " wordend" | 将索引移动到当前索引指向的单词的末尾。单词的定义是一系列字母、数字、下划线或任何非空白字符的组合。注意：使用该表达式前边必须用一个空格隔开 |

**提示**

> 只要结果不产生歧义，关键字可以被缩写，空格也可以省略。例如，"+ 5 chars"可以简写成"+5c"。

在实现中，为了确保表达式为普通字符串，你可以使用 str 或格式化操作来创建一个表达式字符串。下面例子演示了如何删除插入光标前面的一个字符：

```
def backspace(event):
    event.widget.delete("%s-1c" % INSERT, INSERT)
```

## 15.11.2 Marks 用法

Marks(标记)通常是嵌入到 Text 组件文本中的不可见对象。事实上，Marks 是指定字符间的位置，并跟随相应的字符一起移动。Marks 有 INSERT、CURRENT 和 user-defined marks(用户自定义的 Marks)。其中，INSERT 和 CURRENT 是 Tkinter 预定义的特殊 Marks，它们不能够被删除。

INSERT(或"insert")用于指定当前插入光标的位置，Tkinter 会在该位置绘制一个闪烁的光标(因此并不是所有的 Marks 都不可见)。

CURRENT(或"current")用于指定与鼠标坐标最接近的位置。不过，如果你紧按鼠标任何一个按钮，它会直到你松开它才响应。

还可以自定义任意数量的 Marks，Marks 的名字是由普通字符串组成，可以是除了空白字

符外的任何字符(为了避免歧义,你应该起一个有意义的名字)。使用 mark_set()方法创建和移动 Marks。

如果在一个 Mark 标记的位置之前插入或删除文本,那么 Mark 跟着一并移动。删除 Marks 需要使用 mark_unset()方法,删除 Mark 周围的文本并不会删除 Mark 本身。

例 1:Mark 事实上就是索引,用于表示位置:

```
text.insert(INSERT, "I love FishC")
text.mark_set("here", "1.2")
text.insert("here", "插")
```

程序实现如图 15-41 所示。

例 2:如果 Mark 前边的内容发生改变,那么 Mark 的位置也会跟着移动(实际上,就是 Mark 会"记住"它后边的"那家伙"):

```
text.insert(INSERT, "I love FishC")
text.mark_set("here", "1.2")
text.insert("here", "插")
text.insert("here", "入")
```

程序实现如图 15-42 所示。




图 15-41　Text 组件(六)

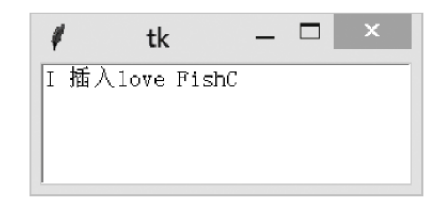



图 15-42　Text 组件(七)

例 3:如果 Mark 周围的文本被删除了,Mark 仍然还在:

```
text.insert(INSERT, "I love FishC")
text.mark_set("here", "1.2")
text.insert("here", "插")
text.delete("1.0", END)
text.insert("here", "入")
```

程序实现如图 15-43 所示。

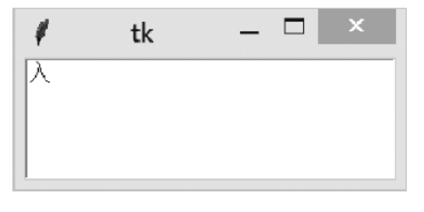



图 15-43　Text 组件(八)

例 4:只有 mark_unset()方法可以解除 Mark 的封印:

```
text.insert(INSERT, "I love FishC")
text.mark_set("here", "1.2")
text.insert("here", "插")
text.mark_unset("here")

text.delete("1.0", END)
text.insert("here", "入")
```

程序实现如图 15-44 所示。

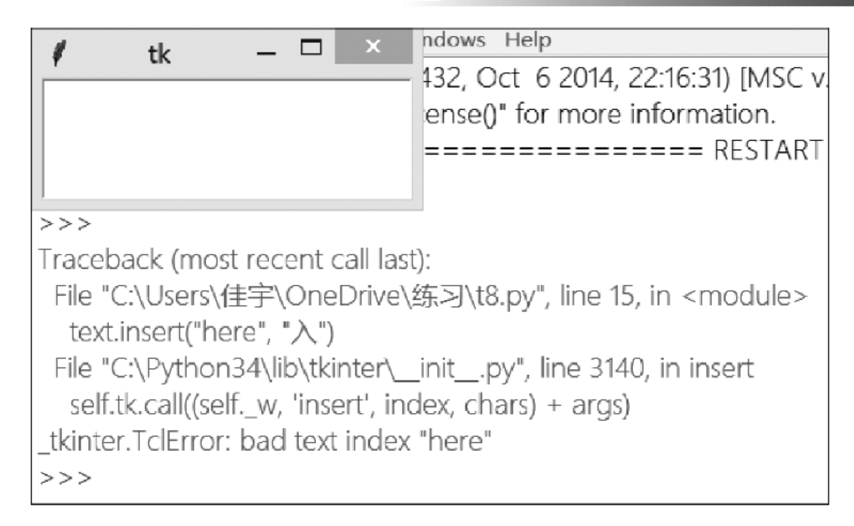



图 15-44 Text 组件(九)

默认插入内容到 Mark，是插入到它的左侧(就是说插入一个字符的话，Mark 向后移动了一个字符的位置)。那么能不能插入到 Mark 的右侧呢？其实是可以的，通过 mark_gravity()方法就可以实现。

例 5(对比例 2)：

```
text.insert(INSERT, "I love FishC")
text.mark_set("here", "1.2")
text.mark_gravity("here", LEFT)
text.insert("here", "插")
text.insert("here", "入")
```

程序实现如图 15-45 所示。

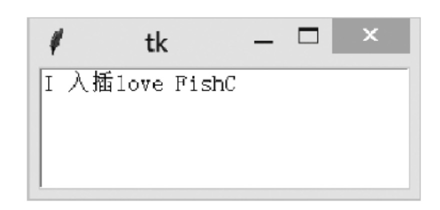



图 15-45 Text 组件(十)

## 15.11.3 Tags 用法

Tags(标签)通常用于改变 Text 组件中内容的样式和功能。可以用来修改文本的字体、尺寸和颜色。另外，Tags 还允许将文本、嵌入的组件和图片与键盘和鼠标等事件相关联。除了 user-defined tags(用户自定义的 Tags)，还有一个预定义的特殊 Tag：SEL。

SEL(或"sel")用于表示对应的选中内容(如果有的话)。

可以自定义任意数量的 Tags，Tags 的名字是由普通字符串组成，可以是除了空白字符外的任何字符。另外，任何文本内容都支持多个 Tags 描述，任何 Tags 也可以用于描述多个不同的文本内容。

为指定文本添加 Tags 可以使用 tag_add()方法：

```
# p15_26.py
from tkinter import *

root = Tk()
text = Text(root, width=30, height=5)
text.pack()
text.insert(INSERT, "I love FishC.com!")
text.tag_add("tag1", "1.7", "1.12", "1.14")
```

```
text.tag_config("tag1", background="yellow", foreground="red")

mainloop()
```




图 15-46　Text 组件(十一)

程序实现如图 15-46 所示。

如上，使用 tag_config()方法可以设置 Tags 的样式。表 15-4 列举了 tag_congif()方法可以使用的选项。

**表 15-4　tag_config()方法可以使用的选项**

| 选　　项 | 含　　义 |
|---|---|
| background | 指定该 Tag 所描述的内容的背景颜色。注意：bg 并不是该选项的缩写，在这里 bg 被解释为 bgstipple 选项的缩写 |
| bgstipple | 指定一个位图作为背景，并使用 background 选项指定的颜色填充。只有设置了 background 选项该选项才会生效。默认的标准位图有 'error'、'gray75'、'gray50'、'gray25'、'gray12'、'hourglass'、'info'、'questhead'、'question'和'warning' |
| borderwidth | 指定文本框的宽度，默认值是 0，只有设置了 relief 选项该选项才会生效。注意：该选项不能使用 bd 缩写 |
| fgstipple | 指定一个位图作为前景色，默认的标准位图有'error'、'gray75'、'gray50'、'gray25'、'gray12'、'hourglass'、'info'、'questhead'、'question'和'warning' |
| font | 指定该 Tag 所描述的内容使用的字体 |
| foreground | 指定该 Tag 所描述的内容的前景色。注意：fg 并不是该选项的缩写，在这里 fg 被解释为 fgstipple 选项的缩写 |
| justify | 控制文本的对齐方式，默认是 LEFT(左对齐)，还可以选择 RIGHT(右对齐)和 CENTER(居中)。注意：需要将 Tag 指向该行的第一个字符，该选项才能生效 |
| lmargin1 | 设置 Tag 指向的文本块第一行的缩进，默认值是 0。注意：需要将 Tag 指向该文本块的第一个字符或整个文本块，该选项才能生效 |
| lmargin2 | 设置 Tag 指向的文本块除了第一行其他行的缩进，默认值是 0。注意：需要将 Tag 指向整个文本块，该选项才能生效 |
| offset | 设置 Tag 指向的文本相对于基线的偏移距离。可以控制文本相对于基线是升高(正数值)或者降低(负数值)，默认值是 0 |
| overstrike | 在 Tag 指定的文本范围画一条删除线，默认值是 False。 |
| relief | 指定 Tag 对应范围的文本的边框样式。可以使用的值有：SUNKEN，RAISED，GROOVE，RIDGE 或 FLAT，默认值是 FLAT(没有边框) |
| rmargin | 设置 Tag 指向的文本块右侧的缩进，默认值是 0 |
| spacing1 | 设置 Tag 所描述的文本块中每一行与上方的空白间隔，默认值是 0。注意：自动换行不算 |
| spacing2 | 设置 Tag 所描述的文本块中自动换行的各行间的空白间隔，默认值是 0。注意：换行符('\n')不算 |
| spacing3 | 设置 Tag 所描述的文本块中每一行与下方的空白间隔，默认值是 0。注意：自动换行不算 |
| tabs | 定制 Tag 所描述的文本块中 Tab 按键的功能，默认 Tab 被定义为 8 个字符的宽度<br>还可以定义多个制表位：tabs=('3c', '5c', '12c')，表示前 3 个 Tab 宽度分别为 3 厘米、5 厘米、12 厘米，接着的 Tab 按照最后两个的差值计算，即 19 厘米、26 厘米、33 厘米<br>注意，'c'的含义是“厘米”而不是“字符”，还可以选择的单位有 'i'(英寸)、'm'(毫米)和 'p'(DPI，大约是'1i'等于'72p')如果是一个整型值，则单位是像素 |
| underline | 若该选项设置为 True 的话，则 Tag 所描述的范围内文本将被加上下划线。默认值是 False |
| wrap | 设置当一行文本的长度超过 width 选项设置的宽度时，是否自动换行。该选项的值可以是 NONE(不自动换行)、CHAR(按字符自动换行)和 WORD(按单词自动换行) |

如果对同一个范围内的文本加上多个 Tags，并且设置相同的选项，那么新创建的 Tag 样式会覆盖比较旧的 Tag：

```
text.tag_config("tag1", background = "yellow", foreground = "red") # 旧的 Tag
text.tag_config("tag2", foreground = "blue") # 新的 Tag
# 那么新创建的 Tag2 会覆盖比较旧的 Tag1 的相同选项
# 注意，与下边的调用顺序没有关系
text.insert(INSERT, "I love FishC.com!", ("tag2", "tag1"))
```

程序实现如图 15-47 所示。

可以使用 tag_raise()和 tag_lower()方法来提高和降低某个 Tag 的优先级：

```
text.tag_config("tag1", background = "yellow", foreground = "red")
text.tag_config("tag2", foreground = "blue")
text.tag_lower("tag2")
text.insert(INSERT, "I love FishC.com!", ("tag2", "tag1"))
```

程序实现如图 15-48 所示。



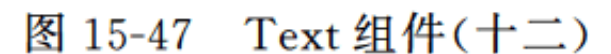


图 15-47 Text 组件(十二)




图 15-48 Text 组件(十三)

Tags 还支持事件绑定，绑定事件使用的是 tag_bind()的方法。下面例子将文本("FishC.com")与鼠标事件进行绑定，当鼠标进入该文本段的时候，鼠标样式切换为"arrow"形态，离开文本段的时候切换回"xterm"形态。当触发鼠标"左键单击操作"事件的时候，使用默认浏览器打开鱼 C 工作室的首页(www.fishc.com)：

```
# p15_27.py
from tkinter import *
import webbrowser

root = Tk()
text = Text(root, width = 30, height = 5)
text.pack()
text.insert(INSERT, "I love FishC.com!")
text.tag_add("link", "1.7", "1.16")
text.tag_config("link", foreground = "blue", underline = True)

def show_hand_cursor(event):
    text.config(cursor = "arrow")

def show_arrow_cursor(event):
    text.config(cursor = "xterm")

def click(event):
    webbrowser.open("http://www.fishc.com")
```

```
text.tag_bind("link", "<Enter>", show_hand_cursor)
text.tag_bind("link", "<Leave>", show_arrow_cursor)
text.tag_bind("link", "<Button-1>", click)

mainloop()
```

程序实现如图 15-49 所示。




图 15-49　Text 组件(十四)

最后,给大家介绍几个 Text 组件使用上的技巧,非常实用。

第一个是判断内容是否发生变化,比如做一个记事本程序,当用户关闭的时候,程序应该检查内容是否有改变,如果有变化,应该提醒用户保存。在下面的例子中,通过校检 Text 组件中文本的 MD5 摘要来判断内容是否发生改变:

```
# p15_28.py
from tkinter import *
import hashlib

root = Tk()
text = Text(root, width=20, height=5)
text.pack()
text.insert(INSERT, "I love FishC.com!")
contents = text.get(1.0, END)

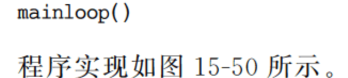
def getSig(contents):
    m = hashlib.md5(contents.encode())
    return m.digest()

sig = getSig(contents)

def check():
    contents = text.get(1.0, END)
    if sig != getSig(contents):
        print("警报:内容发生变动!")
    else:
        print("风平浪静~")

Button(root, text="检查", command=check).pack()

mainloop()
```



程序实现如图 15-50 所示。

第二个例子是查找操作,使用 search()方法可以搜索 Text 组件中的内容。可以提供一个确切的目标进行搜索(默认),也可以使用 Tcl 格式的正则表达式进行搜索(需设置 regexp 选项为 True):

```
# p15_29.py
from tkinter import *
```






图 15-50　Text 组件(十五)

```
root = Tk()
text = Text(root, width=30, height=5)
text.pack()
text.insert(INSERT, "I love FishC.com!")

# 将任何格式的索引号统一为元组 (行,列) 的格式输出
def getIndex(text, index):
    return tuple(map(int, str.split(text.index(index), ".")))

start = 1.0
while True:
    pos = text.search("o", start, stopindex=END)
    if not pos:
        break
    print("找到啦,位置是: ", getIndex(text, pos))
    start = pos + "+1c" # 将 start 指向下一个字符

mainloop()
```

程序实现如图 15-51 所示。

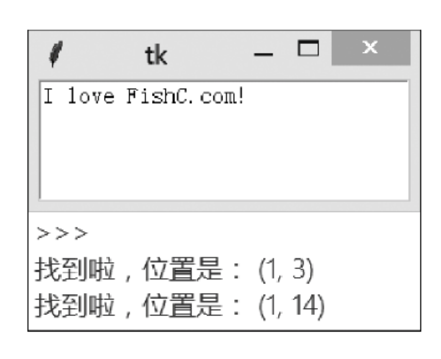



图 15-51　Text 组件(十六)

如果忽略 stopindex 选项，表示直到文本的末尾结束搜索。设置 backwards 选项为 True，则是修改搜索的方向(变为向后搜索，那么 start 变量应该设置为 END，stopindex 选项设置为 1.0，最后"+1c"改为"−1c")。

最后，Text 组件还支持“恢复”和“撤销”操作，这使得 Text 组件显得相当高大上。通过设置 undo 选项为 True 可以开启 Text 组件的“撤销”功能，然后用 edit_undo()方法实现“撤销”操作，用 edit_redo()方法实现“恢复”操作。

```
# p15_30.py
from tkinter import *

root = Tk()
text = Text(root, width=30, height=5, undo=True)
text.pack()
text.insert(INSERT, "I love FishC")
```

```
def show():
    text.edit_undo()

Button(root, text = "撤销", command = show).pack()

mainloop()
```

这是因为 Text 组件内部有一个栈专门用于记录内容的每次变动，所以每次“撤销”操作就是一次弹栈操作，“恢复”就是再次压栈。如图 15-52 所示。

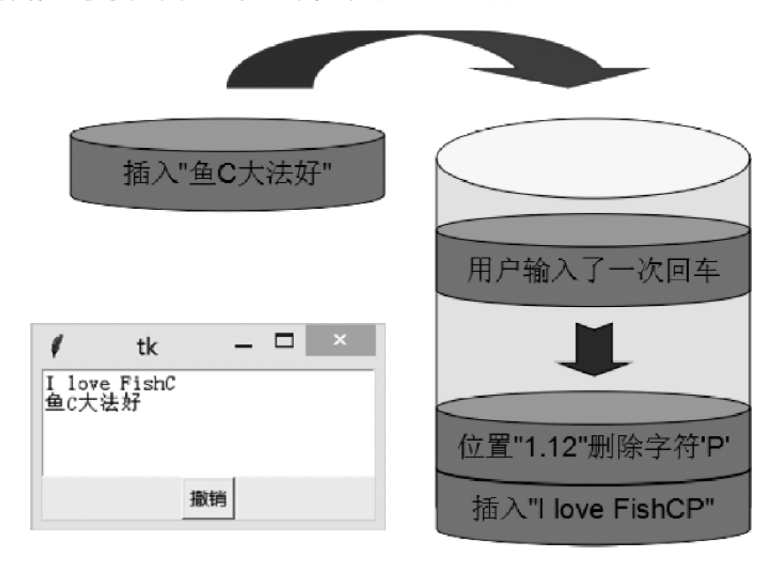



图 15-52　Text 组件(十七)

默认情况下，每一次完整的操作都会放入栈中。但怎么样算是一次完整的操作呢？Tkinter 觉得每次焦点切换、用户按下回车键、删除/插入操作的转换等之前的操作算是一次完整的操作。也就是说，你连续输入“FishC 是个 P”的话，一次“撤销”操作就会将所有的内容删除。

那能不能自定义呢？比如希望插入一个字符就算一次完整的操作，然后每次单击“撤销”就去掉一个字符。

当然可以！做法就是先将 autoseparators 选项设置为 False(因为这个选项是让 Tkinter 在认为一次完整的操作结束后自动插入“分隔符”)，然后绑定键盘事件，每次有输入就用 edit_separator()方法人为地插入一个“分隔符”：

```
# p15_31.py
from tkinter import *

root = Tk()
text = Text(root, width = 30, height = 5, autoseparators = False, undo = True, maxundo = 10)
text.pack()

def callback(event):
    text.edit_separator()
```

```
text.bind('<Key>', callback)
text.insert(INSERT, "I love FishC")

def show():
    text.edit_undo()

Button(root, text = "撤销", command = show).pack()

mainloop()
```

## 15.12　Canvas 组件



虽然我们能用 Tkinter 设计不少东西了，但我知道肯定还是有不少读者感觉对界面编程的“掌控”还不够。说白了，就是还没法随心所欲地去绘制我们想要的界面。

Canvas 组件，是一个可以让你任性的组件，一个可以让你随心所欲地绘制界面的组件。Canvas 是一个通用的组件，它通常用于显示和编辑图形。可以用它来绘制直线、圆形、多边形，甚至是绘制其他组件。

在 Canvas 组件上绘制对象，可以用 create_xxx()的方法（xxx 表示对象类型，例如直线 line、矩形 rectangle 和文本 text 等）：

```
# p15_32.py
from tkinter import *

root = Tk()
w = Canvas(root, width = 200, height = 100)
w.pack()
# 画一条黄色的横线
w.create_line(0, 50, 200, 50, fill = "yellow")
# 画一条红色的竖线(虚线)
w.create_line(100, 0, 100, 100, fill = "red", dash = (4, 4))
# 中间画一个蓝色的矩形
w.create_rectangle(50, 25, 150, 75, fill = "blue")

mainloop()
```

程序实现如图 15-53 所示。

图 15-53　Canvas 组件(一)

注意，添加到 Canvas 上的对象会一直保留着。如果你希望修改它们，你可以使用 coords()、itemconfig()和 move()方法来移动画布上的对象，或者使用 delete()方法来删除：

```
# p15_33.py
...
line1 = w.create_line(0, 50, 200, 50, fill = "yellow")
line2 = w.create_line(100, 0, 100, 100, fill = "red", dash = (4, 4))
rect1 = w.create_rectangle(50, 25, 150, 75, fill = "blue")
w.coords(line1, 0, 25, 200, 25)
w.itemconfig(rect1, fill = "red")
w.delete(line2)
```

```
Button(root, text = "删除全部", command = (lambda x = ALL : w.delete(x))).pack()
…
```

程序实现如图 15-54 所示。

还可以在 Canvas 上显示文本，使用的是 create_text()方法：

```
# p15_34.py
…
w.create_line(0, 0, 200, 100, fill = "green", width = 3)
w.create_line(200, 0, 0, 100, fill = "green", width = 3)
w.create_rectangle(40, 20, 160, 80, fill = "green")
w.create_rectangle(65, 35, 135, 65, fill = "yellow")
w.create_text(100, 50, text = "FishC")
…
```

程序实现如图 15-55 所示。




图 15-54　Canvas 组件(二)

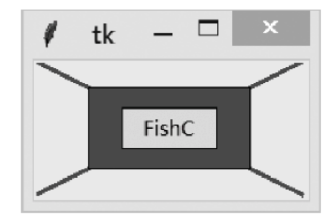



图 15-55　Canvas 组件(三)

使用 create_oval()方法绘制椭圆形(或圆形)，参数是指定一个限定矩形(Tkinter 会自动在这个矩形内绘制一个椭圆)：

```
# p15_35.py
…
w.create_rectangle(40, 20, 160, 80, dash = (4, 4))
w.create_oval(40, 20, 160, 80, fill = "pink")
w.create_text(100, 50, text = "FishC")
…
```

程序实现如图 15-56 所示。

而绘制圆形就是把限定矩形设置为正方形即可：

```
w.create_oval(70, 20, 130, 80, fill = "pink")
```

程序实现如图 15-57 所示。

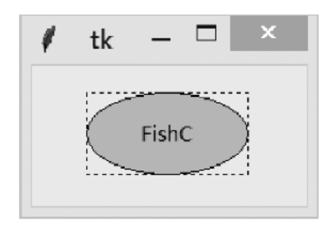



图 15-56　Canvas 组件(四)

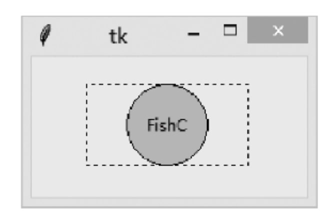



图 15-57　Canvas 组件(五)

如果想要绘制多边形，可以使用 create_polygon()方法。好，现在带大家来画一个五角星。首先，要先确定五个角的坐标。那么高中数学不是体育老师教的鱼油们应该看得懂这张图（看不懂也没关系哈，知道结果就行，因为现在的目标是用 Tkinter 画五角星，而不是学三角函数），如图 15-58 所示。

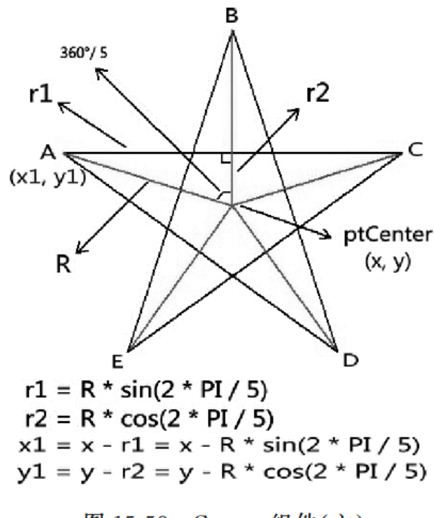



图 15-58　Canvas 组件（六）

```
# p15_36.py
from tkinter import *
import math as m

root = Tk()
w = Canvas(root, width = 200, height = 100, background = "red")
w.pack()
center_x = 100
center_y = 50
r = 50
points = [
    # 左上点
    center_x - int(r * m.sin(2 * m.pi / 5)),
    center_y - int(r * m.cos(2 * m.pi / 5)),
    # 右上点
    center_x + int(r * m.sin(2 * m.pi / 5)),
    center_y - int(r * m.cos(2 * m.pi / 5)),
    # 左下点
    center_x - int(r * m.sin(m.pi / 5)),
    center_y + int(r * m.cos(m.pi / 5)),
    # 顶点
    center_x,
    center_y - r,
    # 右下点
    center_x + int(r * m.sin(m.pi / 5)),
    center_y + int(r * m.cos(m.pi / 5)),
    ]
w.create_polygon(points, outline = "green", fill = "yellow")
```

```
mainloop()
```

程序实现如图 15-59 所示。




图 15-59 Canvas 组件(七)

接着设计一个像 Windows 画图工具那样的面板，让用户可以在上面随心所欲地绘画，如图 15-60 所示。

其实实现原理也很简单，就是获取用户拖动鼠标的坐标，然后每个坐标对应绘制一个点上去就可以了。在这里，不得不承认有点遗憾让人的是 Tkinter 并没有提供画“点”的方法。

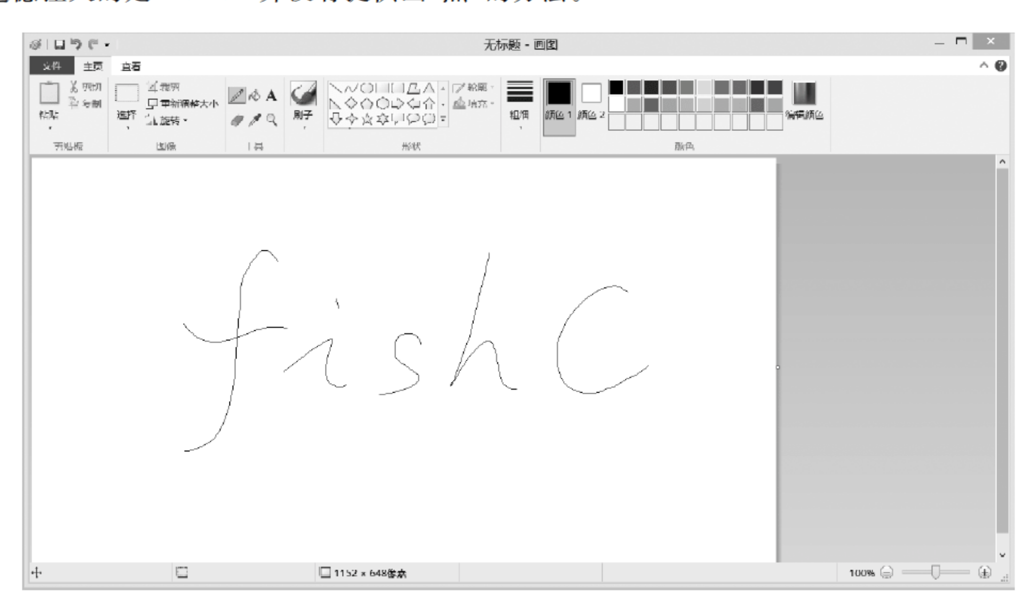



图 15-60 Canvas 组件(八)

但是程序是死的，程序员是活的！可以通过绘制一个超小的椭圆形来表示一个“点”。在下面的例子中，通过响应“鼠标左键按住拖动”事件(<B1-Motion>)，在鼠标拖动的同时获取鼠标的实时位置(x, y)，并绘制一个超小的椭圆来代表一个“点”：

```
# p15_37.py
from tkinter import *

root = Tk()
w = Canvas(root, width = 400, height = 200)
w.pack()

def paint(event):
    x1, y1 = (event.x - 1), (event.y - 1)
    x2, y2 = (event.x + 1), (event.y + 1)
    w.create_oval(x1, y1, x2, y2, fill = "red")

w.bind("<B1 - Motion>", paint)
Label(root, text = "按住鼠标左键并移动，开始绘制你的理想蓝图吧......").pack(side = BOTTOM)

mainloop()
```

程序实现如图 15-61 所示。






图 15-61　Canvas 组件(九)

关于画布对象还有些概念，我们觉得必须了解，这里给大家做个总结：

### 1. Canvas 组件支持的对象。

- arc(弧形、弦或扇形)。
- bitmap(内建的位图文件或 XBM 格式的文件)。
- image(BitmapImage 或 PhotoImage 的实例对象)。
- line(线)。
- oval(圆或椭圆形)。
- polygon(多边形)。
- rectangle(矩形)。
- text(文本)。
- window(组件)。

其中，弦、扇形、椭圆形、圆形、多边形和矩形这些"封闭式"图形都是由轮廓线和填充颜色组成的，通过 outline 和 fill 选项设置它们的颜色，还可以设置为透明(传入空字符串表示透明)。

### 2. 坐标系

由于画布可能比窗口大(带有滚动条的 Canvas 组件)，因此 Canvas 组件可以选择使用两种坐标系：

- 窗口坐标系——以窗口的左上角作为坐标原点。
- 画布坐标系——以画布的左上角作为坐标原点。

### 3. 画布对象显示的顺序

Canvas 组件中创建的画布对象都会被列入显示列表中，越接近背景的画布对象位于显示列表的越下方。显示列表决定当两个画布对象重叠的时候是如何覆盖的(默认情况下新创建的会覆盖旧的画布对象的重叠部分，即位于显示列表上方的画布对象将覆盖下方那个)。当然，显示列表中的画布对象可以被重新排序。

### 4. 指定画布对象

Canvas 组件提供几种方法可以指定画布对象：

- Item handles。
- Tags。
- ALL。
- CURRENT。

Item handles 事实上是一个用于指定某个画布对象的整型数字(也称为画布对象的 ID)。当你在 Canvas 组件上创建一个画布对象的时候,Tkinter 将自动为其指定一个在该 Canvas 组件中独一无二的整型值,然后各种 Canvas 的方法可以通过这个值操纵该画布对象。

Tags 是附在画布对象上的标签,Tags 由普通的非空白字符串组成。一个画布对象可以与多个 Tags 相关联,一个 Tags 也可用于描述多个画布对象。然而,与 Text 组件不同,没有指定画布对象的 Tags 不能进行事件绑定和配置样式。也就是说,Canvas 组件的 Tags 是仅为画布对象所拥有。

Canvas 组件预定义了两个 Tags:ALL 和 CURRENT。

- ALL(或 all)表示 Canvas 组件中的所有画布对象。
- CURRENT(或 current)表示鼠标指针下的画布对象(如果有的话)。

## 15.13 Menu 组件



几乎每个应用程序都可以看到菜单,而常见的菜单有"文件"、"编辑"、"帮助",打开"文件"之后,它会下拉出若干菜单项,例如"新建"、"打开"、"保存"、"退出"等,如图 15-62 所示。




图 15-62 Menu 组件(一)

Tkinter 提供了一个 Menu 组件,用于实现顶级菜单、下拉菜单和弹出菜单。由于该组件是底层代码实现和优化,所以不建议你自行通过按钮和其他组件来实现菜单功能。

创建一个顶级菜单,需要先创建一个菜单实例,然后使用 add()方法将命令和其他子菜单添加进去:

```
# p15_38.py
from tkinter import *

root = Tk()
def callback():
    print("~被调用了~")

# 创建一个顶级菜单
menubar = Menu(root)
```

```
menubar.add_command(label = "Hello", command = callback)
menubar.add_command(label = "Quit", command = root.quit)

# 显示菜单
root.config(menu = menubar)

mainloop()
```

程序实现如图 15-63 所示。

创建一个下拉菜单(或者其他子菜单)，方法也是大同小异，最主要的区别是它们最后需要添加到主菜单上(而不是窗口上)：

```
# p15_39.py
from tkinter import *

root = Tk()

def callback():
    print("～被调用了～")

# 创建一个顶级菜单
menubar = Menu(root)
# 创建一个下拉菜单"文件",然后将它添加到顶级菜单中
filemenu = Menu(menubar, tearoff = False)
filemenu.add_command(label = "打开", command = callback)
filemenu.add_command(label = "保存", command = callback)
filemenu.add_separator()
filemenu.add_command(label = "退出", command = root.quit)
menubar.add_cascade(label = "文件", menu = filemenu)
# 创建另一个下拉菜单"编辑",然后将它添加到顶级菜单中
editmenu = Menu(menubar, tearoff = False)
editmenu.add_command(label = "剪切", command = callback)
editmenu.add_command(label = "拷贝", command = callback)
editmenu.add_command(label = "粘贴", command = callback)
menubar.add_cascade(label = "编辑", menu = editmenu)
# 显示菜单
root.config(menu = menubar)

mainloop()
```

程序实现如图 15-64 所示。

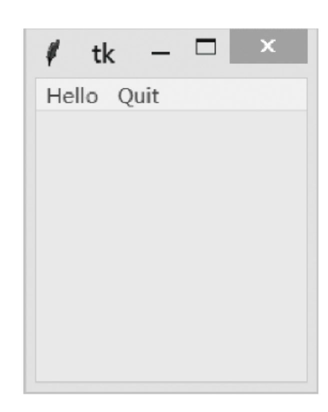



图 15-63 Menu 组件(二)




图 15-64 Menu 组件(三)

创建一个弹出菜单方法也是一致的,不过需要使用 post()方法明确地将其显示出来:

```
# p15_40.py
from tkinter import *

root = Tk()

def callback():
    print("~被调用了~")

# 创建一个弹出菜单
menu = Menu(root, tearoff=False)
menu.add_command(label="撤销", command=callback)
menu.add_command(label="重做", command=callback)
frame = Frame(root, width=512, height=512)
frame.pack()

def popup(event):
    menu.post(event.x_root, event.y_root)

# 绑定鼠标右键
frame.bind("<Button-3>", popup)

mainloop()
```

大家发现在创建一个 Menu 组件的时候都把一个叫 tearoff 的选项设置为 False。那么这个翻译为"撕开"的选项有什么用呢?Tkinter 要我们撕开什么呢?好了,大家不要想歪了,试试便知——把 tearoff 改为 True 之后,"文件"菜单增加了一行小横杠,如图 15-65 所示。

单击一下,噢,原来 Tkinter 让我们打开的是菜单,如图 15-66 所示。




图 15-65 Menu 组件(四)




图 15-66 Menu 组件(五)

最后,这个菜单不仅可以添加常见的命令菜单项,还可以添加单选按钮或多选按钮,那么用法就跟 Checkbutton 组件和 Radiobutton 组件类似啦。

```
# p15_41.py
from tkinter import *

root = Tk()
```

```
def callback():
    print("~被调用了~")

# 创建一个顶级菜单
menubar = Menu(root)
# 创建 checkbutton 关联变量
openVar = IntVar()
saveVar = IntVar()
exitVar = IntVar()
# 创建一个下拉菜单"文件",然后将它添加到顶级菜单中
filemenu = Menu(menubar, tearoff=True)
filemenu.add_checkbutton(label="打开", command=callback, variable=openVar)
filemenu.add_checkbutton(label="保存", command=callback, variable=saveVar)
filemenu.add_separator()
filemenu.add_checkbutton(label="退出", command=root.quit, variable=exitVar)
menubar.add_cascade(label="文件", menu=filemenu)
# 创建 radiobutton 关联变量
editVar = IntVar()
editVar.set(1)
# 创建另一个下拉菜单"编辑",然后将它添加到顶级菜单中
editmenu = Menu(menubar, tearoff=True)
editmenu.add_radiobutton(label="剪切", command=callback, 
variable=editVar, value=1)
editmenu.add_radiobutton(label="拷贝", command=callback, 
variable=editVar, value=2)
editmenu.add_radiobutton(label="粘贴", command=callback, 
variable=editVar, value=3)
menubar.add_cascade(label="编辑", menu=editmenu)
# 显示菜单
root.config(menu=menubar)

mainloop()
```

图 15-67　Menu 组件(六)

程序实现如图 15-67 所示。

## 15.14　Menubutton 组件

Menubutton 组件是一个与 Menu 组件相关联的按钮,它可以放在窗口中的任意位置,并且在被按下时弹出下拉菜单。这个组件是有一定的历史意义的,在 Tkinter 的早期版本,使用 Menubutton 组件来实现顶级菜单,但现在直接用 Menu 组件就可以实现了。因此,现在该组件适用于你希望菜单按钮出现在其他位置的时候。

创建一个 Menubutton 组件,并创建一个 Menu 组件与之相关联:

```
# p15_42.py
from tkinter import *

root = Tk()

def callback():
    print("~被调用了~")
```

```
mb = Menubutton(root, text = "点我", relief = RAISED)
mb.pack()
filemenu = Menu(mb, tearoff = False)
filemenu.add_checkbutton(label = "打开", command = callback, selectcolor = "yellow")
filemenu.add_command(label = "保存", command = callback)
filemenu.add_separator()
filemenu.add_command(label = "退出", command = root.quit)
mb.config(menu = filemenu)

mainloop()
```

程序实现如图 15-68 所示。




图 15-68　Menubutton 组件

## 15.15　OptionMenu 组件

OptionMenu(选项菜单)事实上是下拉菜单的改版,它的发明弥补了 Listbox 组件无法实现下拉列表框的遗憾。创建一个选择菜单非常简单,只需要它一个 Tkinter 变量(用于记录用户选择了什么)以及若干选项即可:

```
# p15_43.py
from tkinter import *

root = Tk()
variable = StringVar()
variable.set("one")
w = OptionMenu(root, variable, "one", "two", "three")
w.pack()

mainloop()
```

程序实现如图 15-69 所示。

要获得用户选择的内容,使用 Tkinter 变量的 get()方法即可:

```
…
def callback():
```

```
    print(variable.get())

Button(root, text = "点我", command = callback).pack()
…
```

修改后程序实现如图 15-70 所示。

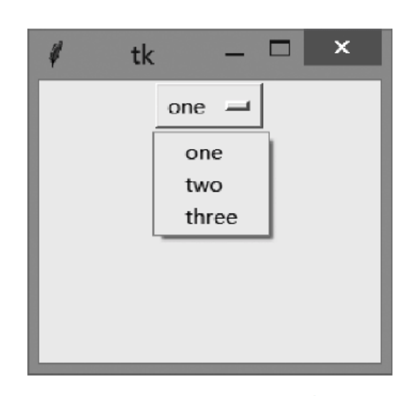



图 15-69　OptionMenu 组件(一)

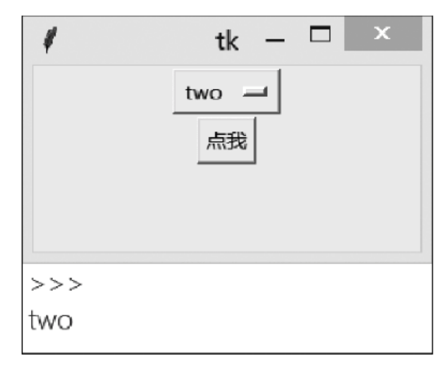



图 15-70　OptionMenu 组件(二)

最后演示如何将很多选项添加到选项菜单中：

```
# p15_44.py
from tkinter import *

OPTIONS = [
    "California",
    "458",
    "FF",
    "ENZO",
    "LaFerrari"
    ]
root = Tk()
variable = StringVar()
variable.set(OPTIONS[0])
w = OptionMenu(root, variable, *OPTIONS)
w.pack()

def callback():
    print(variable.get())

Button(root, text = "点我", command = callback).pack()

mainloop()
```

程序实现如图 15-71 所示。




图 15-71　OptionMenu 组件(三)

## 15.16　Message 组件



Message(消息)组件是 Label 组件的变体,用于显示多行文本消息。Message 组件能够自动换行,并调整文本的尺寸使其适应给定的尺寸。

```
# p15_45.py
from tkinter import *

root = Tk()
w1 = Message(root, text = "这是一则消息", width = 100)
w1.pack()
w2 = Message(root, text = "这是一则骇人听闻的长长长长长消息!", width = 100)
w2.pack()

mainloop()
```

程序实现如图 15-72 所示。




图 15-72　Message 组件

## 15.17　Spinbox 组件

Spinbox 组件(Tk8.4 新增)是 Entry 组件的变体,用于从一些固定的值中选取一个。Spinbox 组件跟 Entry 组件用法非常相似,主要区别是使用 Spinbox 组件,可以通过范围或者元组指定允许用户输入的内容。

```
# p15_46.py
from tkinter import *

root = Tk()
w = Spinbox(root, from_ = 0, to = 10)
w.pack()

mainloop()
```

程序实现如图 15-73 所示。

还可以通过元组指定允许输入的值:

```
…
w = Spinbox(root, values = ("小甲鱼", "~风介~", "wei_Y", "戴宇轩"))
…
```

程序修改后实现如图 15-74 所示。

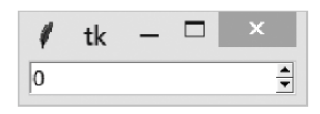



图 15-73　Spinbox 组件(一)

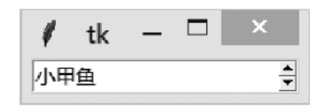



图 15-74　Spinbox 组件(二)

## 15.18　PanedWindow 组件

PanedWindow 组件(Tk8.4 新增)是一个空间管理组件。跟 Frame 组件类似,都是为组件提供一个框架,不过 PanedWindow 允许让用户调整应用程序的空间划分。

创建一个两窗格的PanedWindow组件非常简单：

```
# p15_47.py
from tkinter import *

m = PanedWindow(orient = VERTICAL)
m.pack(fill = BOTH, expand = 1)
top = Label(m, text = "top pane")
m.add(top)
bottom = Label(m, text = "bottom pane")
m.add(bottom)

mainloop()
```

程序实现如图15-75所示。

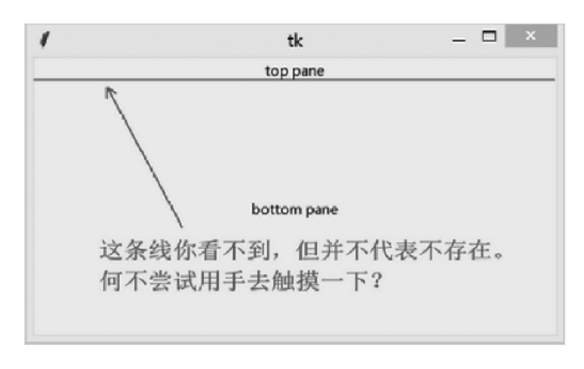



图15-75 PanedWindow组件(一)

创建一个三窗格的PanedWindow组件则需要一点小技巧：

```
# p15_48.py
from tkinter import *

m1 = PanedWindow()
m1.pack(fill = BOTH, expand = 1)
left = Label(m1, text = "left pane")
m1.add(left)
m2 = PanedWindow(orient = VERTICAL)
m1.add(m2)
top = Label(m2, text = "top pane")
m2.add(top)
bottom = Label(m2, text = "bottom pane")
m2.add(bottom)

mainloop()
```

程序实现如图15-76所示。

这里不同窗格事实上是有一条“分割线”(sash)隔开，虽然你看不到，但你却可以感受到它的存在。不妨把鼠标缓慢移动到大概的位置，当鼠标指针改变的时候后拖动鼠标……也可以把“分割线”显式地显示出来，并且可以为它附上一个“手柄”(handle)：

```
# p15_49.py
```

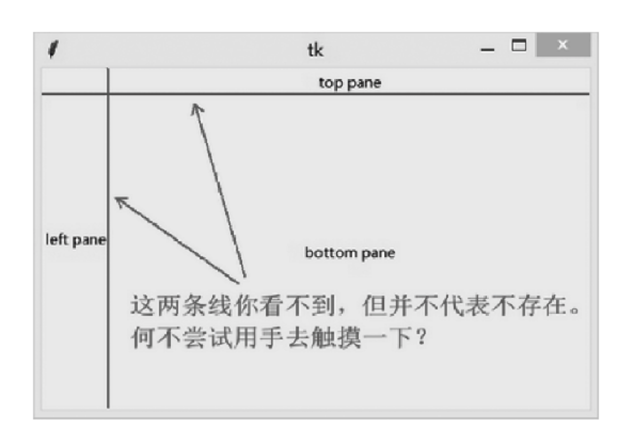



图 15-76　PanedWindow 组件(二)

```
from tkinter import *

m1 = PanedWindow(showhandle = True, sashrelief = SUNKEN)
m1.pack(fill = BOTH, expand = 1)
left = Label(m1, text = "left pane")
m1.add(left)
m2 = PanedWindow(orient = VERTICAL, showhandle = True, sashrelief = SUNKEN)
m1.add(m2)
top = Label(m2, text = "top pane")
m2.add(top)
bottom = Label(m2, text = "bottom pane")
m2.add(bottom)

mainloop()
```

程序实现如图 15-77 所示。

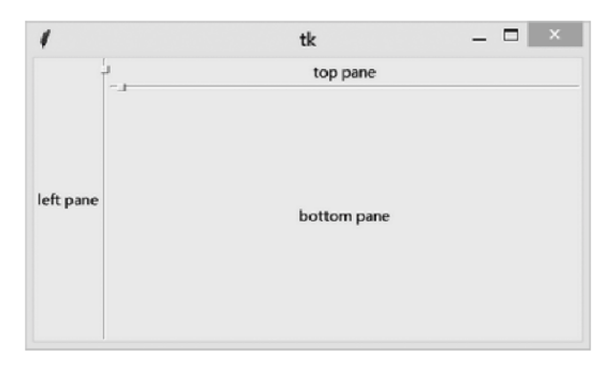



图 15-77　PanedWindow 组件(三)

## 15.19　Toplevel 组件

Toplevel(顶级窗口)组件类似于 Frame 组件，但 Toplevel 组件是一个独立的顶级窗口，这种窗口通常拥有标题栏、边框等部件。Toplevel 组件通常用在显示额外的窗口、对话框和其

他弹出窗口中。

在下面的例子中，在 root 窗口添加一个按钮用于创建一个顶级窗口，点一下出现一个：

```
# p15_50.py
from tkinter import *

root = Tk()

def create():
    top = Toplevel()
    top.title("FishC Demo")
    msg = Message(top, text = "I love FishC.com")
    msg.pack()

Button(root, text = "创建顶级窗口", command = create).pack()

mainloop()
```

程序实现如图 15-78 所示。




图 15-78　Toplevel 组件(一)

想要几个就点几下，如图 15-79 所示。

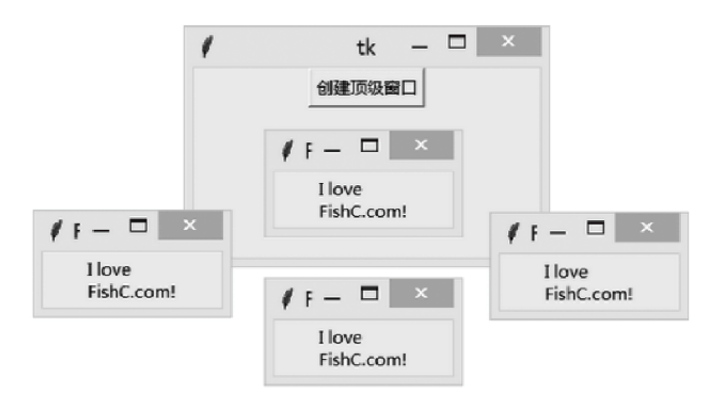



图 15-79　Toplevel 组件(二)

最后，Tkinter 提供这一系列方法用于与窗口管理器进行交互。它们可以被 Tk(根窗口)进行调用，同样也适用于 Toplevel(顶级窗口)。详情请查看 http://bbs.fishc.com/thread-61246-1-1.html。

这里有必要讲一下的是 attributes()这个方法,它用于设置和获取窗口属性,如果只给出选项名,将返回当前窗口该选项的值。注意:以下选项不支持关键字参数,需要在选项前添加横杠(-)并用字符串的方式表示,用逗号(,)隔开选项和值。

下面演示将 Toplevel 的窗口设置为 50%透明:

```
# p15_51.py
from tkinter import *

root = Tk()

def create():
    top = Toplevel()
    top.title("FishC Demo")
    top.attributes("-alpha", 0.5)
    msg = Message(top, text="I love FishC.com")
    msg.pack()

Button(root, text="创建顶级窗口", command=create).pack()

mainloop()
```

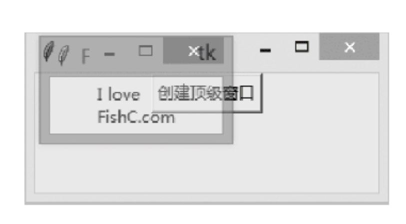



图 15-80 Toplevel 组件(三)

程序实现如图 15-80 所示。

## 15.20 事件绑定



一个 Tkinter 应用程序大部分时间花费在事件循环中(通过 mainloop()方法进入)。事件可以有各种来源,包括用户触发的鼠标、键盘操作和窗口管理器触发的重绘事件(在多数情况下是由用户间接引起的)。

Tkinter 提供一个强大的机制可以让你自由地处理事件,对于每个组件来说,可以通过 bind()方法将函数或方法绑定到具体的事件上。当被触发的事件满足该组件绑定的事件时,Tkinter 就会带着事件描述去调用 handler()方法。

下面有几个例子,请随意感受下:

```
# p15_45.py
# 捕获单击鼠标的位置
from tkinter import *

root = Tk()

def callback(event):
    print("点击位置:", event.x, event.y)

frame = Frame(root, width=200, height=200)
frame.bind("<Button-1>", callback)
frame.pack()

mainloop()
```

程序实现如图 15-81 所示。

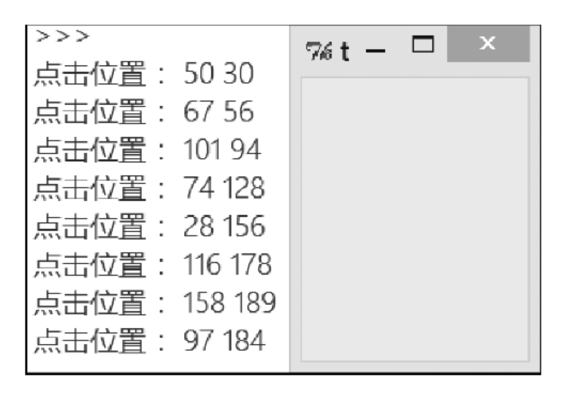



图 15-81 事件绑定(一)

在上面这个例子中，使用 Frame 组件的 bind()方法将鼠标单击事件(<Button-1>)和自定义的 callback()方法绑定起来。那么运行后的结果是——当你单击鼠标左键的时候，IDLE 会相应地将鼠标的位置显示出来。

只有当组件获得焦点的时候才能接收键盘事件(Key)，下面的例子中用 focus_set()获得焦点，你可以设置 Frame 的 takefocus 选项为 True，然后使用 Tab 将焦点转移上来。

```
# p15_46.py
# 捕获键盘事件
from tkinter import *

root = Tk()

def callback(event):
    print("敲击位置：", repr(event.char))

frame = Frame(root, width=200, height=200)
frame.bind("<Key>", callback)
frame.focus_set()
frame.pack()

mainloop()
```

程序实现如图 15-82 所示。

最后一个例子展示捕获鼠标在组件上的运动轨迹，这里需要关注的是<Motion>事件：

```
# p15_47.py
from tkinter import *

root = Tk()

def callback(event):
    print("当前位置：", event.x, event.y)

frame = Frame(root, width=200, height=200)
```




图 15-82　事件绑定（二）

```
frame.bind("<Motion>", callback)
frame.pack()

mainloop()
```

## 15.21　事件序列

Tkinter 使用一种称为事件序列的机制来允许用户定义事件，用户需使用 bind()方法将具体的事件序列与自定义的方法绑定。事件序列是以字符串的形式表示的，可以表示一个或多个相关联的事件（如果是多个事件，那么对应的方法只有在满足所有事件的前提下才会被调用）。

事件序列使用以下语法描述：

```
<modifier-type-detail>
```

- 事件序列是包含在尖括号（<...>）中。
- type 部分的内容是最重要的，它通常用于描述普通的事件类型，例如鼠标单击或键盘按键单击（详见表 15-5）。
- modifier 部分的内容是可选的，它通常用于描述组合键，例如 Ctrl＋C、Shift＋鼠标左键单击（详见表 15-6）。
- detail 部分的内容是可选的，它通常用于描述具体的按键，例如 Button-1 表示鼠标左键。比如：

(1) <Button-1>表示用户单击鼠标左键。

(2) <KeyPress-H>表示用户按下 H 键。

(3) <Control-Shift-KeyPress-H>表示用户同时按下 Ctrl＋Shift＋H 键。

### 15.21.1　type

表 15-5 列举了 type 部分常用的关键词及含义。

表 15-5 type 部分常用的关键词及含义

| type | 含义 |
|---|---|
| Activate | 当组件的状态从“未激活”变为“激活”的时候触发该事件 |
| Button | 当用户单击鼠标按键的时候触发该事件。detail 部分指定具体哪个按键：<Button-1>鼠标左键，<Button-2>鼠标中键，<Button-3>鼠标右键，<Button-4>滚轮上滚(Linux)，<Button-5>滚轮下滚(Linux) |
| ButtonRelease | 当用户释放鼠标按键的时候触发该事件。在大多数情况下，比 Button 要更好用，因为如果当用户不小心按下鼠标，用户可以将鼠标移出组件再释放鼠标，从而避免不小心触发事件 |
| Configure | 当组件的尺寸发生改变的时候触发该事件 |
| Deactivate | 当组件的状态从“激活”变为“未激活”的时候触发该事件 |
| Destroy | 当组件被销毁的时候触发该事件 |
| Enter | 当鼠标指针进入组件的时候触发该事件。注意：不是指用户按下回车键 |
| Expose | 当窗口或组件的某部分不再被覆盖的时候触发该事件 |
| FocusIn | 当组件获得焦点的时候触发该事件。用户可以用 Tab 键将焦点转移到该组件上(需要该组件的 takefocus 选项为 True)，也可以调用 focus_set()方法使该组件获得焦点 |
| FocusOut | 当组件失去焦点的时候触发该事件 |
| KeyPress | 当用户按下键盘按键的时候触发该事件。detail 可以指定具体的按键，例如<KeyPress-H>表示当大写字母 H 被按下的时候触发该事件。KeyPress 可以简写为 Key |
| KeyRelease | 当用户释放键盘按键的时候触发该事件 |
| Leave | 当鼠标指针离开组件的时候触发该事件 |
| Map | 当组件被映射的时候触发该事件。意思是在应用程序中显示该组件的时候，例如，调用 grid()方法 |
| Motion | 当鼠标在组件内移动的整个过程均触发该事件 |
| MouseWheel | 当鼠标滚轮滚动的时候触发该事件。目前该事件仅支持 Windows 和 Mac 系统，Linux 系统请参考 Button |
| Unmap | 当组件被取消映射的时候触发该事件。意思是在应用程序中不再显示该组件的时候，例如调用 grid_remove()方法 |
| Visibility | 当应用程序至少有一部分在屏幕中是可见的时候触发该事件 |

## 15.21.2 modifier

表 15-6 列举了 modifier 部分常用的关键词及含义。

表 15-6 modifier 部分常用的关键词及含义

| modifier | 含义 |
|---|---|
| Alt | 当按下 Alt 按键的时候 |
| Any | 表示任何类型的按键被按下的时候。例如，<Any-KeyPress>表示当用户按下任何按键时触发事件 |
| Control | 当按下 Ctrl 按键的时候 |
| Double | 当后续两个事件被连续触发的时候。例如<Double-Button-1>表示当用户双击鼠标左键时触发事件 |
| Lock | 当打开大写字母锁定键(CapsLock)的时候 |
| Shift | 当按下 Shift 按键的时候 |
| Triple | 跟 Double 类似，当后续三个事件被连续触发的时候 |

## 15.22 Event 对象

当 Tkinter 去回调预先定义的函数时，将带着 Event 对象(作为参数)去调用，表 15-7 列举了 Event 对象的属性及含义。

**表 15-7 Event 对象的属性及含义**

| 属　性 | 含　义 |
| --- | --- |
| widget | 产生该事件的组件 |
| x，y | 当前的鼠标位置坐标(相对于窗口左上角，像素为单位) |
| x_root，y_root | 当前的鼠标位置坐标(相对于屏幕左上角，像素为单位) |
| char | 按键对应的字符(键盘事件专属) |
| keysym | 按键名，见下方 Key names(键盘事件专属) |
| keycode | 按键码，见下方 Key names(键盘事件专属) |
| num | 按钮数字(鼠标事件专属) |
| width，height | 组件的新尺寸(Configure 事件专属) |
| type | 该事件类型 |

当事件为<Key>、<KeyPress>、<KeyRelease>的时候，detail 可以通过设定具体的按键名(keysym)来筛选。例如<Key-H>表示按下键盘上的大写字母 H 时候触发事件，<Key-Tab>表示按下键盘上的 Tab 按键的时候触发事件。

表 15-8 列举了键盘所有特殊按键的 keysym 和 keycode(其中的按键码是对应美国标准 101 键盘的 Latin-1 字符集，键盘标准不同对应的按键码不同，但按键名是一样的)。

**表 15-8 键盘所有特殊按键的 keysym 和 keycode**

| 按键名(keysym) | 按键码(keycode) | 代表的按键 |
| --- | --- | --- |
| Alt_L | 64 | 左边的 Alt 按键 |
| Alt_R | 113 | 右边的 Alt 按键 |
| BackSpace | 22 | Backspace(退格)按键 |
| Cancel | 110 | break 按键 |
| Caps_Lock | 66 | CapsLock(大写字母锁定)按键 |
| Control_L | 37 | 左边的 Ctrl 按键 |
| Control_R | 109 | 右边的 Ctrl 按键 |
| Delete | 107 | Delete 按键 |
| Down | 104 | ↓按键 |
| End | 103 | End 按键 |
| Escape | 9 | Esc 按键 |
| Execute | 111 | SysReq 按键 |
| F1 | 67 | F1 按键 |
| F2 | 68 | F2 按键 |
| F3 | 69 | F3 按键 |
| F4 | 70 | F4 按键 |
| F5 | 71 | F5 按键 |
| F6 | 72 | F6 按键 |

续表

| 按键名(keysym) | 按键码(keycode) | 代表的按键 |
|---|---|---|
| F7 | 73 | F7 按键 |
| F8 | 74 | F8 按键 |
| F9 | 75 | F9 按键 |
| F10 | 76 | F10 按键 |
| F11 | 77 | F11 按键 |
| F12 | 96 | F12 按键 |
| Home | 97 | Home 按键 |
| Insert | 106 | Insert 按键 |
| Left | 100 | ←按键 |
| Linefeed | 54 | Linefeed(Ctrl ＋ J) |
| KP_0 | 90 | 小键盘数字 0 |
| KP_1 | 87 | 小键盘数字 1 |
| KP_2 | 88 | 小键盘数字 2 |
| KP_3 | 89 | 小键盘数字 3 |
| KP_4 | 83 | 小键盘数字 4 |
| KP_5 | 84 | 小键盘数字 5 |
| KP_6 | 85 | 小键盘数字 6 |
| KP_7 | 79 | 小键盘数字 7 |
| KP_8 | 80 | 小键盘数字 8 |
| KP_9 | 81 | 小键盘数字 9 |
| KP_Add | 86 | 小键盘的＋按键 |
| KP_Begin | 84 | 小键盘的中间按键(5) |
| KP_Decimal | 91 | 小键盘的点按键(.) |
| KP_Delete | 91 | 小键盘的删除键 |
| KP_Divide | 112 | 小键盘的/按键 |
| KP_Down | 88 | 小键盘的↓按键 |
| KP_End | 87 | 小键盘的 End 按键 |
| KP_Enter | 108 | 小键盘的 Enter 按键 |
| KP_Home | 79 | 小键盘的 Home 按键 |
| KP_Insert | 90 | 小键盘的 Insert 按键 |
| KP_Left | 83 | 小键盘的←按键 |
| KP_Multiply | 63 | 小键盘的 * 按键 |
| KP_Next | 89 | 小键盘的 PageDown 按键 |
| KP_Prior | 81 | 小键盘的 PageUp 按键 |
| KP_Right | 85 | 小键盘的→按键 |
| KP_Subtract | 82 | 小键盘的-按键 |
| KP_Up | 80 | 小键盘的↑按键 |
| Next | 105 | PageDown 按键 |
| Num_Lock | 77 | NumLock(数字锁定)按键 |
| Pause | 110 | Pause(暂停)按键 |
| Print | 111 | PrintScrn(打印屏幕)按键 |

续表

| 按键名(keysym) | 按键码(keycode) | 代表的按键 |
|---|---|---|
| Prior | 99 | PageUp 按键 |
| Return | 36 | Enter(回车)按键 |
| Right | 102 | →按键 |
| Scroll_Lock | 78 | ScrollLock 按键 |
| Shift_L | 50 | 左边的 Shift 按键 |
| Shift_R | 62 | 右边的 Shift 按键 |
| Tab | 23 | Tab(制表)按键 |
| Up | 98 | ↑按键 |

## 15.23 布局管理器



什么是布局管理器？说白了就是管理你的那些组件如何排列的家伙。Tkinter 有三个布局管理器，分别是 pack、grid 和 place，其中：

- pack 是按添加顺序排列组件。
- grid 是按行/列形式排列组件。
- place 允许程序员指定组件的大小和位置。

### 15.23.1 pack

pack 其实之前的例子一直在用，对比 grid 管理器，pack 更适用于少量组件的排列，但它在使用上更加简单。如果需要创建相对复杂的布局结构，那么建议是使用多个框架(Frame)结构，或者使用 grid 管理器实现。

注意

> 不要在同一个父组件中混合使用 pack 和 grid，因为 Tkinter 会很认真地在那儿计算到底先使用哪个布局管理器……以至于你等了半个小时，Tkinter 还在那儿纠结不出结果！

我们常常会遇到的一个情况是将一个组件放到一个容器组件中，并填充整个父组件。下面生成一个 Listbox 组件并将它填充到 root 窗口中：

```
# p15_55.py
from tkinter import *

root = Tk()
listbox = Listbox(root)
listbox.pack(fill = BOTH, expand = True)
for i in range(10):
    listbox.insert(END, str(i))

mainloop()
```

程序实现如图 15-83 所示。

其中，fill 选项是告诉窗口管理器该组件将填充整个分配给它的空间，BOTH 表示同时横向和纵向扩展，X 表示横向，Y 表示纵向；expand 选项是告诉窗口管理器将父组件的额外空间也填满。

默认情况下，pack 是将添加的组件依次纵向排列：

```
# p15_56.py
from tkinter import *

root = Tk()
Label(root, text = "Red", bg = "red", fg = "white").pack(fill = X)
Label(root, text = "Green", bg = "green", fg = "black").pack(fill = X)
Label(root, text = "Blue", bg = "blue", fg = "white").pack(fill = X)

mainloop()
```

程序实现如图 15-84 所示。

如果想要组件横向挨个儿排列，可以使用 side 选项：

```
…
Label(root, text = "Red", bg = "red", fg = "white").pack(side = LEFT)
Label(root, text = "Green", bg = "green", fg = "black").pack(side = LEFT)
Label(root, text = "Blue", bg = "blue", fg = "white").pack(side = LEFT)
…
```

程序修改后，实现如图 15-85 所示。




图 15-83 pack 管理器

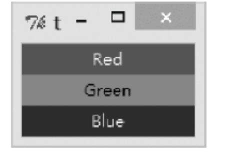



图 15-84 纵向排列

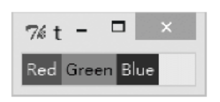



图 15-85 横向排列

## 15.23.2 grid

grid 管理器可以说是 Tkinter 这三个布局管理器中最灵活多变的。当你在设计对话框的时候，使用 gird 尤其便捷。如果你此前一直在用 pack 构造窗口布局，那么学习完 grid 你会悔恨当初为啥不早学它。使用一个 grid 就可以简单地实现你用很多个框架和 pack 搭建起来的效果。

使用 grid 排列组件，只需告诉它你想要将组件放置的位置（行/列，row 选项指定行，cloumn 选项指定列）。此外，你并不用提前指出网格（grid 分布给组件的位置称为网格）的尺寸，因为管理器会自动计算。

```
# p15_57.py
from tkinter import *

root = Tk()
# column 默认值是 0
Label(root, text = "用户名").grid(row = 0)
Label(root, text = "密码").grid(row = 1)
Entry(root).grid(row = 0, column = 1)
Entry(root, show = " * ").grid(row = 1, column = 1)

mainloop()
```

程序实现如图 15-86 所示。

默认情况下组件会居中显示在对应的网格里，你可以使用 sticky 选项来修改这一特性。该选项可以使用的值有 E、W、S、N(EWSN 分别表示东西南北，即上北下南左西右东)以及它们的组合。因此，可以通过 sticky = W 使得 Label 左对齐：

```
…
Label(root, text = "用户名").grid(row = 0, sticky = W)
Label(root, text = "密码").grid(row = 1, sticky = W)
…
```

程序修改后，实现如图 15-87 所示。

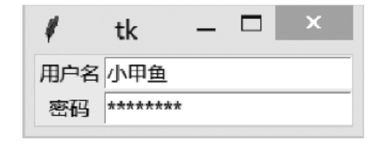



图 15-86　grid 管理器

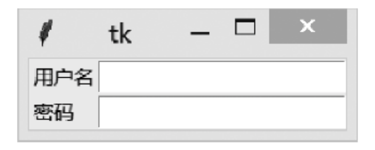



图 15-87　sticky 选项修改对齐方式

有时候可能需要用几个网格来放置一个组件，可以做到吗？当然可以，只需要指定 rowspan 和 columnspan 就可以实现跨行和跨列的功能：

```
# p15_58.py
from tkinter import *

root = Tk()
Label(root, text = "用户名").grid(row = 0, sticky = W)
Label(root, text = "密码").grid(row = 1, sticky = W)

Entry(root).grid(row = 0, column = 1)
Entry(root, show = " * ").grid(row = 1, column = 1)
photo = PhotoImage(file = "logo.gif")
Label(root, image = photo).grid(row = 0, column = 2,
rowspan = 2, padx = 5, pady = 5)
Button(text = "提交", width = 10).grid(row = 2,
columnspan = 3, pady = 5)

mainloop()
```




图 15-88　跨行和跨列布局

程序实现如图 15-88 所示。

## 15.23.3 place

通常情况下不建议使用 place 布局管理器，因为对比起 pack 和 grid，place 要做更多的工作。不过纯在即合理，place 在一些特殊的情况下可以发挥妙用。请看下面的例子。

使用 place，可以将子组件显示在父组件的正中间：

```
# p15_59.py
from tkinter import *

root = Tk()

def callback():
    print("正中靶心")

Button(root, text = "点我", command = callback).place(relx = 0.5, rely = 0.5, anchor = CENTER)

mainloop()
```

程序实现如图 15-89 所示。

在某种情况下，或许你希望一个组件可以覆盖另一个组件，那么 place 又可以派上用场了。下面例子演示用 Button 覆盖 Label 组件：

```
# p15_60.py
from tkinter import *

root = Tk()

def callback():
    print("正中靶心")

photo = PhotoImage(file = "logo_big.gif")
Label(root, image = photo).pack()
Button(root, text = "点我", command = callback).place(relx = 0.5, rely = 0.5, anchor = CENTER)

mainloop()
```

程序实现如图 15-90 所示。




图 15-89　place 管理器




图 15-90　利用 place 覆盖组件

不难看出，relx 和 rely 选项指定的是相对于父组件的位置，范围是 00～1.0，因此 0.5 表示位于正中间。那么 relwidth 和 relheight 选项则是指定相对于父组件的尺寸：

```
# p15_61.py
from tkinter import *

root = Tk()
Label(root, bg = "red").place(relx = 0.5, rely = 0.5,
relheight = 0.75, relwidth = 0.75, anchor = CENTER)
Label(root, bg = "yellow").place(relx = 0.5, rely = 0.5,
relheight = 0.5, relwidth = 0.5, anchor = CENTER)
Label(root, bg = "green").place(relx = 0.5, rely = 0.5,
relheight = 0.25, relwidth = 0.25, anchor = CENTER)

mainloop()
```

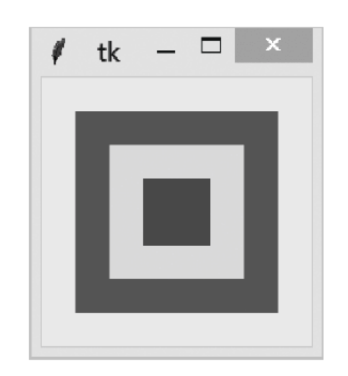


图 15-91　相对位置和相对尺寸

程序实现如图 15-91 所示。

对于上面的代码，无论你如何拉伸改变窗口，三个 Label 的尺寸均会跟着同步。

## 15.24　标准对话框

Tkinter 提供了三种标准对话框模块，分别是：

- messagebox。
- filedialog。
- colorchooser。

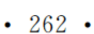
这三个模块原来是独立的，分别是 tkMessageBox、tkFileDialog 和 tkColorChooser，需要导入才能使用。在 Python3 之后，这些模块全部被收归到 tkinter 模块的麾下。下面的所有演示都是在 Python3 下实现的，如果你用的是 Python2.x，请在文件处加入 import tkMessageBox，然后将 messagebox 替换为 tkMessageBox 即可。

### 15.24.1　messagebox（消息对话框）

表 15-9 列举使用 messagebox 可以创建的所有标准对话框样式。

**表 15-9　messagebox 创建的标准对话框样式**

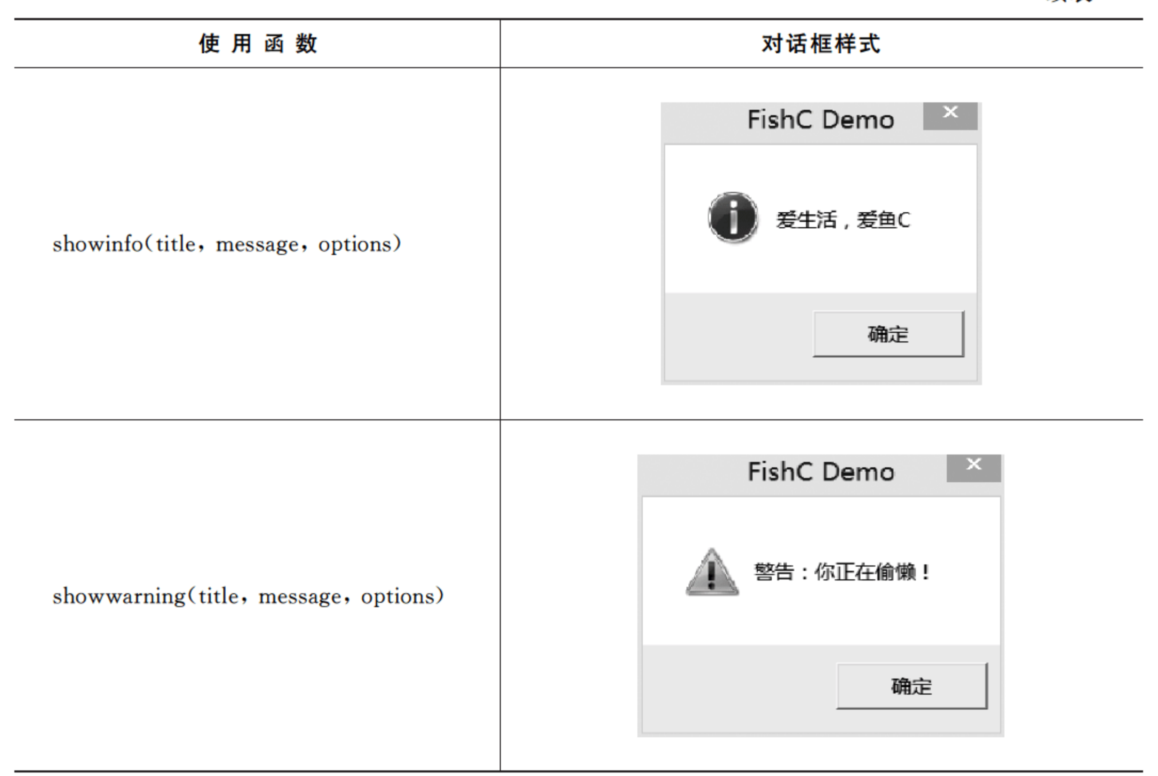
| 使 用 函 数 | 对话框样式 |
|---|---|
| askokcancel(title, message, options) | FishC Demo 发射核弹？ 确定 取消 |



续表

| 使 用 函 数 | 对话框样式 |
| --- | --- |
| askquestion(title，message，options) | FishC Demo<br>买个U盘？<br>是(Y)　否(N) |
| askretrycancel(title，message，options) | FishC Demo<br>启动失败，重试？<br>重试(R)　取消 |
| askyesno(title，message，options) | FishC Demo<br>你是鱼油吗？<br>是(Y)　否(N) |
| showerror(title，message，options) | FishC Demo<br>出错啦~！<br>确定 |

续表



| 使 用 函 数 | 对话框样式 |
| --- | --- |
| showinfo(title, message, options) | FishC Demo<br>爱生活，爱鱼C<br>确定 |
| showwarning(title, message, options) | FishC Demo<br>警告：你正在偷懒！<br>确定 |

### 1. 参数

所有的这些函数都有相同的参数：

- title 参数毋庸置疑是设置标题栏的文本。
- message 参数是设置对话框的主要文本内容，可以用'\n'来实现换行。
- options 参数可以设置的选项和含义如表 15-10 所示。

**表 15-10　options 参数可以设置的选项和含义**

| 选项 | 含　　义 |
| --- | --- |
| default | 1. 设置默认的按钮(也就是按下回车响应的那个按钮)<br>2. 默认是第一个按钮(像“确定”、“是”或“重试”)<br>3. 可以设置的值根据对话框函数的不同可以选择：CANCEL、IGNORE、OK、NO、RETRY 或 YES |
| icon | 1. 指定对话框显示的图标<br>2. 可以指定的值有：ERROR、INFO、QUESTION 或 WARNING<br>3. 注意：不能指定自己的图标 |
| parent | 1. 如果不指定该选项，那么对话框默认显示在根窗口上<br>2. 如果想要将对话框显示在子窗口 w 上，那么可以设置 parent=w |

### 2. 返回值

askokcancel()，askretrycancel()和 askyesno()返回布尔类型的值：

- 返回 True 表示用户单击了“确定”或“是”按钮。
- 返回 False 表示用户单击了“取消”或“否”按钮。

askquestion()返回"yes"或"no"字符串表示用户单击了“是”或“否”按钮。

showerror()，showinfo()和 showwarning()返回"ok"表示用户单击了“是”按钮。

## 15.24.2 filedialog（文件对话框）

当应用程序需要使用打开文件或保存文件的功能时，文件对话框显得尤为重要。实现起来就是这样：

```
# p15_62.py
from tkinter import *

root = Tk()

def callback():
    fileName = filedialog.askopenfilename()
    print(fileName)

Button(root, text = "打开文件", command = callback).pack()

mainloop()
```

程序实现如图 15-92 所示。

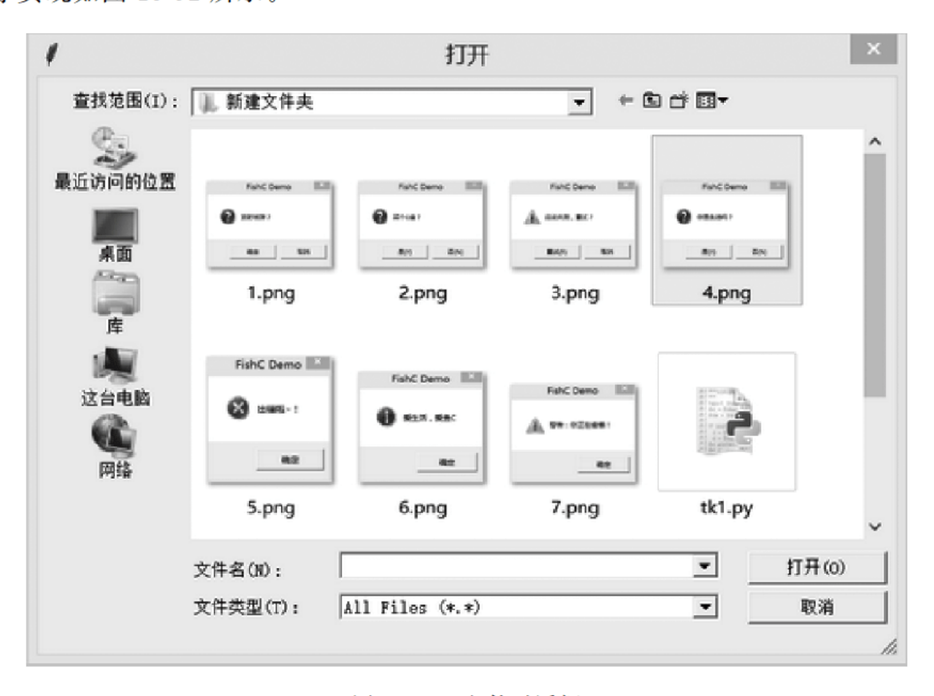



图 15-92 文件对话框

filedialog 模块提供了两个函数：askopenfilename( ** option)和 asksaveasfilename( ** option)，分别用于打开文件和保存文件。

### 1. 参数

两个函数可供设置的选项是一样的，表 15-11 列举了可用的选项及含义。

**表 15-11　可用选项及含义**

| 选　　项 | 含　　义 |
| --- | --- |
| defaultextension | 指定文件的后缀，例如：defaultextension＝".jpg"，那么当用户输入一个文件名"FishC"的时候，文件名会自动添加后缀为"FishC.jpg"。注意：如果用户输入文件名包含后缀，那么该选项不生效 |
| filetypes | 指定筛选文件类型的下拉菜单选项，该选项的值是由 2 元组构成的列表。每个 2 元组由（类型名，后缀）构成，例如，filetypes＝[（"PNG"，".png"），（"JPG"，".jpg"），（"GIF"，".gif"）] |
| initialdir | 指定打开/保存文件的默认路径。默认路径是当前文件夹 |
| parent | 如果不指定该选项，那么对话框默认显示在根窗口上。如果想要将对话框显示在子窗口 w 上，那么可以设置 parent＝w |
| title | 指定文件对话框的标题栏文本 |

### 2. 返回值

- 如果用户选择了一个文件，那么返回值是该文件的完整路径。
- 如果用户单击了取消按钮，那么返回值是空字符串。

## 15.24.3 colorchooser（颜色选择对话框）

颜色选择对话框提供一个让用户选择颜色的界面，请看下面的例子：

```
# p15_63.py
from tkinter import *

root = Tk()

def callback():
    fileName = colorchooser.askcolor()
    print(fileName)

Button(root, text = "选择颜色", command = callback).pack()

mainloop()
```

程序实现如图 15-93 所示。

### 1. 参数

askcolor(color，** option)函数的 color 参数用于指定初始化的颜色，默认是浅灰色；option 参数可以指定的选项及含义如表 15-12 所示。



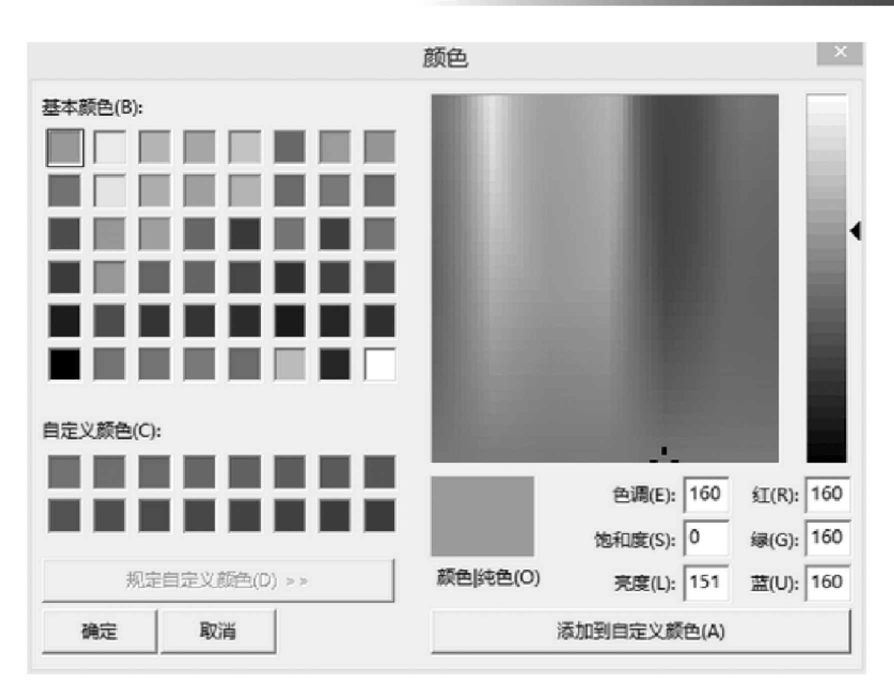



图 15-93　颜色选择对话框

**表 15-12　option 参数可以指定的选项及含义**

| 选项 | 含　　义 |
| --- | --- |
| title | 指定颜色对话框的标题栏文本 |
| parent | 如果不指定该选项，那么对话框默认显示在根窗口上。如果想要将对话框显示在子窗口 w 上，那么可以设置 parent＝w |

### 2. 返回值

- 如果用户选择一个颜色并单击“确定”按钮后，返回值是一个二元组，第 1 个元素是选择的 RGB 颜色值，第 2 个元素是对应的十六进制颜色值。
- 如果用户单击“取消”按钮，那么返回值是(None，None)。

# 第16章 Pygame：游戏开发

## 16.1 安装 Pygame



在 Python 中提到游戏开发，那肯定非 Pygame 莫属了。Pygame 是一个利用 SDL 库实现的模块。(注：SDL(Simple DirectMedia Layer)是一套开放源代码的跨平台多媒体开发库，使用 C 语言写成。SDL 提供了数种控制图像、声音、输出入的函数，让开发者只要用相同或是相似的代码就可以开发出跨多个平台(Linux、Windows、Mac OS X 等)的应用软件。目前 SDL 多用于开发游戏、模拟器、媒体播放器等多媒体应用领域)。

Pygame 官网：www.pygame.org。

我们看到 Pygame 的 LOGO 很形象，是一条蟒蛇叼着一个游戏手柄，如图 16-1 所示。




图 16-1　Pygame 的 LOGO

单击 Downloads 按钮，可以找到对应 Python 版本的 Pygame 模块，如图 16-2 所示。

- pygame-1.9.1.win32-py2.7.msi 3.1MB
- pygame-1.9.1release.win32-py2.4.exe 3MB
- pygame-1.9.1release.win32-py2.5.exe 3MB
- pygame-1.9.1.win32-py2.5.msi 3MB
- pygame-1.9.1.win32-py2.6.msi 3MB
- pygame-1.9.2a0.win32-py2.7.msi 6.4MB
- pygame-1.9.1.win32-py3.1.msi 3MB
- pygame-1.9.2a0.win32-py3.2.msi 6.4MB
- (optional) Numeric for windows python2.5 (note: Numeric is old, best to use numpy) py2.5.exe
- windows 64bit users note: use the 32bit python with this 32bit pygame.

图 16-2　Pygame 的下载地址

这里有几点需要注意：

- pygame-1.9.1 的 1.9.1 是 Pygame 的版本号。

- pygame-1.9.2a0 的 a 表示 alpha 版本（测试版），一般开发周期为：pre-alpha -> alpha -> beta -> release。
- 升级 Pygame 版本需要先卸载旧版本。
- 目前提供的 Pygame 基本上都是 32 位版本，因此需要先安装 32 位的 Python。

本书使用的是 pygame-1.9.2a0.win32-py3.4.msi，安装包在附件中可以找到，下载之后直接默认安装即可。

OK，安装完成后，打开 IDLE：

```
>>> import pygame
>>> print(pygame.ver)
1.9.2a0
```

成功打印版本号，说明安装正确。如果出现"ImportError: DLL load failed: %1 不是有效的 Win32 应用程序。"错误，请检查你的 Python 版本是不是 64 位的。目前提供的 Pygame 是 32 位的，因此 Python 也需要安装对应的 32 位版本。

作为一个游戏模块，Pygame 实现的功能主要有：

- 绘制图形。
- 显示图片。
- 动画效果。
- 与键盘、鼠标和游戏手柄等外设交互。
- 播放声音。
- 碰撞检测。

## 16.2 初步尝试

这是本书最后一个章节，现在对于大家来说，最好的学习方法应该是直接钻进代码里面去：

```
# p16_1/turtle.py
import pygame
import sys

pygame.init()                                           # 初始化 Pygame
size = width, height = 600, 400
speed = [-2, 1]
bg = (255, 255, 255)
screen = pygame.display.set_mode(size)                  # 创建指定大小的窗口
pygame.display.set_caption("初次见面，请大家多多关照！")  # 设置窗口标题
turtle = pygame.image.load("turtle.png")
position = turtle.get_rect()                            # 获得图像的位置矩形

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
    position = position.move(speed)                     # 移动图像
```

```
    if position.left < 0 or position.right > width:
        turtle = pygame.transform.flip(turtle, True, False) # 翻转图像
        speed[0] = - speed[0]                                # 反方向移动
    if position.top < 0 or position.bottom > height:
        speed[1] = - speed[1]
    screen.fill(bg)                                          # 填充背景
    screen.blit(turtle, position)                            # 更新图像
    pygame.display.flip()                                    # 更新界面
    pygame.time.delay(10)                                    # 延迟 10 毫秒
```

程序实现如图 16-3 所示。




图 16-3　第一个 Pygame 游戏

这是一个简单的演示：小乌龟会不断地移动，并且每当移动到窗口的左右边界的位置，还会自动“掉头”。

代码分析：

pygame 其实是一个包，里边包含着很多不同功能的模块。开头的 pygame.init()就是用于初始化这些模块，让它们做好准备，随时待命。

```
screen = pygame.display.set_mode(size)
```

display.set_mode()方法创建一个 Surface 对象，在这里将它作背景画布，后面将它填充为纯白色。

```
turtle = pygame.image.load("turtle.png")
```

image.load()方法用于加载图片，不得不说 Pygame 比 Tkinter 要“厚道”，因为 Pygame 不仅支持 GIF 格式，还支持时下流行的 JPG、PNG、BMP 等格式的图片。

图片成功加载之后，Pygame 会帮你将图片转换为一个 Surface 对象并返回。要让小乌龟移动，事实上就是不断修改这个 Surface 对象的位置。现在问题来了：如何修改？

```
position = turtle.get_rect()
```

get_rect()用于获得该 Surface 对象的矩形区域，其实这个矩形区域也是一个对象，主要用来描述图像的位置大小信息。

紧接着进入一个“死循环”，这是确保游戏可以不断地运行下去。有些读者可能会纳闷了：那怎么关闭程序？

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        sys.exit()
```

学过了界面编程，我们已知道事件和事件循环。Pygame 也是如此，用户的一切行为都会变成一个个事件消息，放入事件队列里边。那么这里就是迭代获取每个事件消息，检测如果是 QUIT(退出)事件，那么就调用 sys.exit()退出程序。

```
    position = position.move(speed)
```

Rect 对象拥有一个 move()方法，用于移动该矩形区域，事实上就是修改该矩形的坐标。

接下来很简单，我们判断移动后的矩形区域是否位于窗口的边界之外，如果出界了，那么要把移动的方向修改一下。

```
turtle = pygame.transform.flip(turtle, True, False)
```

小乌龟每次“撞墙”之后都会“掉头”，主要就是由 transform.flip()方法实现。该方法用于翻转图片，第二个参数表示水平翻转，第三个参数表示垂直翻转。

```
screen.fill(bg)
screen.blit(turtle, position)
```

这两句用于填充背景颜色和将移动后的小乌龟放上去。没错，Surface 对象的 blit()方法就是用于将一个 Surface 对象放到另一个 Surface 对象上方。

```
pygame.display.flip()
```

最后要做的就是刷新画面，由于 Pygame 采用的是双缓冲模式，因此需要调用 display.flip()方法将缓冲好的画面一次性刷新到显示器上。所谓双缓冲，即在内存中创建一个与屏幕绘图区域一致的对象，先将图形绘制到内存中的这个对象上，再一次性将这个对象上的图形复制到屏幕上，这样能大大加快绘图的速度以及避免闪烁现象。

```
pygame.time.delay(10)
```

当这一切都完成之后，调用 time.delay()方法让程序挂起 10 毫秒，这样小乌龟才不会跟发了疯一样到处乱窜。

## 16.3 解惑



### 16.3.1 什么是 Surface 对象

什么是 Surface 对象呢？简单来说 Surface 对象就是 Pygame 用来表示图像的对象。所以以后说图像，就是指 Surface 对象，说 Surface 对象，就是指图像。

### 16.3.2 将一个图像绘制到另一个图像上是怎么一回事

Surface 对象的 blit()方法是将一个图像绘制到另一个图像上，如图 16-4 所示。




图 16-4 blit()方法

上面是两个 Surface 对象，一个是作为背景的白色画布，一个是加载图片并转换得到的小乌龟。那请问，现在在我们面前的是一个图像还是两个图像？

答案是一个！

我们知道图像是由像素组成的，例如我把小乌龟的眼睛放大，大家就可以清楚地看到其实是由一些带颜色的马赛克组成的，而这些马赛克，称为像素，如图 16-5 所示。

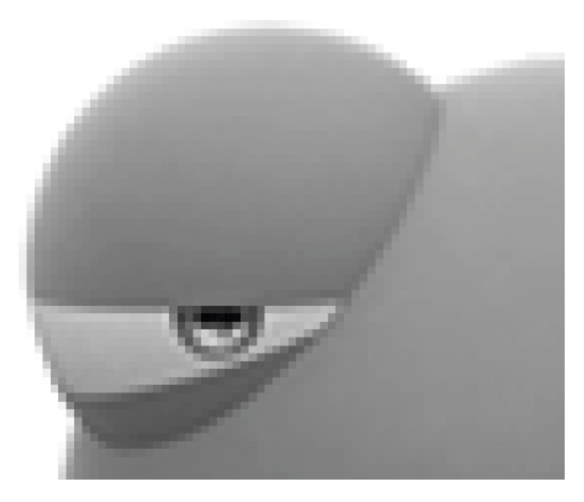


图 16-5 像素

用 blit()方法将一个图像放到另一个图像上，其实并不是真的把一个图像复制上去，事实上 Pygame 只是修改其中一个图像某些位置的像素颜色，从而达到覆盖的效果。

### 16.3.3 移动图像是怎么一回事

图像移动以及移动的快慢涉及帧率问题，在游戏开发和视频制作中我们都经常听到帧这个关键词。帧率就是一秒钟可以切换多少次图像。刚才提到 Pygame 支持40～200 帧，说的就是 Pygame 支持每秒钟切换 40～200 次图像。

那么小乌龟是如何移动的呢？看下面的代码：

```
…
    position = position.move(speed)    # 移动图像
…
    screen.fill(bg)                    # 填充背景
    screen.blit(turtle, position)      # 更新图像
    pygame.display.flip()              # 更新界面
…
```

调用 Rect 对象的 move()方法，事实上就是修改这个矩形范围的位置，例如这里 speed 是[－2, 1]，那么每次调用 move()方法，就相当于水平位置－2，垂直位置＋1 的意思。

位置移动后调用 screen.fill()将整个背景画布刷白，这样位于上一个位置的小乌龟也就被同时刷掉了。然后将当前移动位置后的小乌龟用 blit()方法画上去(事实上就是修改背景画布中小乌龟位置的像素颜色)。最后用 flip()方法将整个修改好的新界面显示出来。而我们讲的帧率，就是指最后 flip()的更新速度。

### 16.3.4 如何控制游戏的速度

由于怕我们的小乌龟乱窜，可以用 time 模块的 delay()方法增加延迟，延迟就是啥都不准动。time 模块其实有个 Clock 类，可以用来实现帧率的控制：

```
…
clock = pygame.time.Clock() # 实例化 Clock 对象

# 创建指定大小的窗口
…
    # pygame.time.delay(10)
    clock.tick(200) # 设置不高于 200 帧执行
```

通过调用 Clock 的 tick()方法来设置帧率，这里将参数设置为 200，表示每秒钟不得超过 200 帧的速度执行。通常用这个方法来控制游戏的速度。不妨可以试试将帧率设置为 1，那么就可以看到一秒钟小乌龟就只移动一下。

### 16.3.5 Pygame 的效率高不高

有读者朋友可能会关心效率问题，因为 Python 虽然简洁好用，但效率不高。而游戏开发对性能有苛刻的追求，例如在复杂的绘制环境中，可以保持越高的帧率，那么游戏体现出来的流畅度就越高。Pygame 里边的大部分模块考虑到效率的原因，都是由 C 语言写成并优化的。因此，效率方面肯定不在话下，官方的数据显示是 40～200 帧每秒执行任何 Pygame 游戏，而一般 30 帧被认为是可以接受的流畅度。

### 16.3.6 我应该从哪里获得帮助

官网(http://www.pygame.org)(可以说 Pygame 的官网已经做得相当不错了)中各种文档、资料、演示代码都很齐全。但很多读者可能看不懂英文文档,国内目前也没有关于 Pygame 文档的翻译计划。于是咱鱼 C 诞生了 Pygame 大文档版块(Pygame 游戏开发帮助文档:http://bbs.fishc.com/forum-323-1.html)。

## 16.4 事件



所谓的游戏,事实上就是一个死循环,如果不去干预它,它就会自己玩得很开心,像前面例子中那个疯狂的小乌龟……而事件,正是 Pygame 提供了干预的机制。例如当用户看烦了小乌龟,可以单击关闭按钮,就会产生 QUIT 事件。代码处理 QUIT 事件的方法就是调用 sys.exit()方法退出程序。

事件随时可能发生(例如用户在窗口上边移动鼠标、单击鼠标、敲击按键等),Pygame 的做法是把所有的事件都存放到事件队列里。通过 for 语句迭代取出每一条事件,然后处理关注的事件即可。

下面的代码将程序运行期间产生的所有事件记录并存放到一个文件中:

```
# p16_2/pg_1.py
import pygame
import sys

pygame.init()
size = width, height = 600, 400
screen = pygame.display.set_mode(size)
pygame.display.set_caption("FishC Demo")
f = open("record.txt", 'w')

while True:
    for event in pygame.event.get():
        f.write(str(event) + '\n')
        if event.type == pygame.QUIT:
            f.close()
            sys.exit()
```

虽然程序停留的时间不长,却产生了不少的事件,如图 16-6 所示。

接下来让这些事件可以"刷刷刷"地显示在画面上,这应该会很酷!那么这就涉及要在屏幕上显示文字的功能,或者说要求我们在 Surface 对象上显示文字。遗憾的是,Pygame 没有办法直接在一个 Surface 对象上面显示文字,因此需要调用 font 模块的 render()方法,该方法是将要显示文字活生生地渲染成一个 Surface 对象,这样就可以调用 blit()方法将一个 Surface 对象放到另一个上面。

```
# p16_2/pg_2.py
import pygame
import sys
```

```
record.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
<Event(17-VideoExpose {})>
<Event(16-VideoResize {'w': 600, 'h': 400, 'size': (600, 400)})>
<Event(1-ActiveEvent {'gain': 0, 'state': 1})>
<Event(4-MouseMotion {'pos': (599, 399), 'buttons': (0, 0, 0), 'rel': (600, 400)})>
<Event(1-ActiveEvent {'gain': 1, 'state': 1})>
<Event(4-MouseMotion {'pos': (587, 336), 'buttons': (0, 0, 0), 'rel': (-12, -63)})>
<Event(4-MouseMotion {'pos': (565, 324), 'buttons': (0, 0, 0), 'rel': (-22, -12)})>
<Event(4-MouseMotion {'pos': (539, 311), 'buttons': (0, 0, 0), 'rel': (-26, -13)})>
<Event(4-MouseMotion {'pos': (520, 301), 'buttons': (0, 0, 0), 'rel': (-19, -10)})>
<Event(4-MouseMotion {'pos': (503, 292), 'buttons': (0, 0, 0), 'rel': (-17, -9)})>
<Event(4-MouseMotion {'pos': (491, 285), 'buttons': (0, 0, 0), 'rel': (-12, -7)})>
<Event(4-MouseMotion {'pos': (480, 280), 'buttons': (0, 0, 0), 'rel': (-11, -5)})>
<Event(4-MouseMotion {'pos': (473, 276), 'buttons': (0, 0, 0), 'rel': (-7, -4)})>
<Event(4-MouseMotion {'pos': (469, 273), 'buttons': (0, 0, 0), 'rel': (-4, -3)})>
<Event(4-MouseMotion {'pos': (468, 272), 'buttons': (0, 0, 0), 'rel': (-1, -1)})>
<Event(4-MouseMotion {'pos': (467, 271), 'buttons': (0, 0, 0), 'rel': (-1, -1)})>
<Event(4-MouseMotion {'pos': (467, 270), 'buttons': (0, 0, 0), 'rel': (0, -1)})>
<Event(4-MouseMotion {'pos': (467, 269), 'buttons': (0, 0, 0), 'rel': (0, -1)})>
<Event(2-KeyDown {'key': 97, 'scancode': 30, 'unicode': '', 'mod': 0})>
<Event(4-MouseMotion {'pos': (468, 269), 'buttons': (0, 0, 0), 'rel': (1, 0)})>
<Event(4-MouseMotion {'pos': (469, 269), 'buttons': (0, 0, 0), 'rel': (1, 0)})>
<Event(4-MouseMotion {'pos': (470, 268), 'buttons': (0, 0, 0), 'rel': (1, -1)})>
<Event(4-MouseMotion {'pos': (471, 267), 'buttons': (0, 0, 0), 'rel': (1, -1)})>
<Event(4-MouseMotion {'pos': (472, 266), 'buttons': (0, 0, 0), 'rel': (1, -1)})>
<Event(3-KeyUp {'scancode': 30, 'key': 97, 'mod': 0})>
<Event(4-MouseMotion {'pos': (474, 265), 'buttons': (0, 0, 0), 'rel': (2, -1)})>
<Event(4-MouseMotion {'pos': (476, 263), 'buttons': (0, 0, 0), 'rel': (2, -2)})>
<Event(4-MouseMotion {'pos': (477, 262), 'buttons': (0, 0, 0), 'rel': (1, -1)})>
<Event(4-MouseMotion {'pos': (479, 260), 'buttons': (0, 0, 0), 'rel': (2, -2)})>
<Event(4-MouseMotion {'pos': (481, 260), 'buttons': (0, 0, 0), 'rel': (2, 0)})>
<Event(4-MouseMotion {'pos': (482, 260), 'buttons': (0, 0, 0), 'rel': (1, 0)})>
<Event(4-MouseMotion {'pos': (483, 260), 'buttons': (0, 0, 0), 'rel': (1, 0)})>
<Event(2-KeyDown {'key': 98, 'scancode': 48, 'unicode': '', 'mod': 0})>
<Event(4-MouseMotion {'pos': (495, 254), 'buttons': (0, 0, 0), 'rel': (12, -6)})>
<Event(4-MouseMotion {'pos': (487, 250), 'buttons': (0, 0, 0), 'rel': (-8, -4)})>
<Event(4-MouseMotion {'pos': (486, 249), 'buttons': (0, 0, 0), 'rel': (-1, -1)})>
<Event(3-KeyUp {'scancode': 48, 'key': 98, 'mod': 0})>
<Event(4-MouseMotion {'pos': (486, 248), 'buttons': (0, 0, 0), 'rel': (0, -1)})>
<Event(4-MouseMotion {'pos': (486, 247), 'buttons': (0, 0, 0), 'rel': (0, -1)})>
<Event(2-KeyDown {'key': 99, 'scancode': 46, 'unicode': '', 'mod': 0})>
<Event(4-MouseMotion {'pos': (487, 247), 'buttons': (0, 0, 0), 'rel': (1, 0)})>
<Event(3-KeyUp {'scancode': 46, 'key': 99, 'mod': 0})>
<Event(2-KeyDown {'key': 100, 'scancode': 32, 'unicode': '', 'mod': 0})>
<Event(3-KeyUp {'scancode': 32, 'key': 100, 'mod': 0})>
<Event(4-MouseMotion {'pos': (487, 248), 'buttons': (0, 0, 0), 'rel': (0, 1)})>
<Event(4-MouseMotion {'pos': (487, 249), 'buttons': (0, 0, 0), 'rel': (0, 1)})>
```

图 16-6 事件

```
pygame.init()
size = width, height = 600, 400
bg = (0, 0, 0)
screen = pygame.display.set_mode(size)
pygame.display.set_caption("FishC Demo")
event_texts = []
# 要在 Pygame 中使用文本,必须创建 Font 对象
# 第一个参数指定字体,第二个参数指定字体的尺寸
font = pygame.font.Font(None, 20)
# 调用 get_linesize()方法获得每行文本的高度
line_height = font.get_linesize()
position = 0
screen.fill(bg)

while True:
    for event in pygame.event.get():
```

```
            if event.type == pygame.QUIT:
                sys.exit()
            # render()方法将文本渲染成 Surface 对象
            # 第一个参数是待渲染的文本
            # 第二个参数指定是否消除锯齿
            # 第三个参数指定文本的颜色
            screen.blit(font.render(str(event), True, (0, 255, 0)), (0, position))
            position += line_height
            if position > height:
                # 满屏时清屏
                position = 0
                screen.fill(bg)
        pygame.display.flip()
```

表 16-1 列举了 Pygame 常用的事件及含义。

**表 16-1 Pygame 常用的事件及含义**

| 事　件 | 含　义 | 属　性 |
|---|---|---|
| QUIT | 按下关闭按钮 | none |
| ATIVEEVENT | Pygame 被激活或者隐藏 | gain，state |
| KEYDOWN | 键盘按键被按下 | unicode，key，mod |
| KEYUP | 键盘按键被松开 | key，mod |
| MOUSEMOTION | 鼠标移动 | pos，rel，buttons |
| MOUSEBUTTONDOWN | 鼠标按键被按下 | pos，button |
| MOUSEBUTTONUP | 鼠标按键被松开 | pos，button |
| JOYAXISMOTION | 游戏手柄上的摇杆移动 | joy，axis，value |
| JOYBALLMOTION | 游戏手柄上的轨迹球滚动 | joy，axis，value |
| JOYHATMOTION | 游戏手柄上的帽子开关移动 | joy，axis，value |
| JOYBUTTONDOWN | 游戏手柄按钮被按下 | joy，button |
| JOYBUTTONUP | 游戏手柄按钮被松开 | joy，button |
| VIDEORESIZE | 用户调整窗口的尺寸 | size，w，h |
| VIDEOEXPOSE | 部分窗口需要重新绘制 | none |
| USEREVENT | 用户定义的事件 | code |

既然已经知道了这么多，想让疯狂的小乌龟受控制应该也不是什么难事了吧？

```
# p16_2/pg_3.py
import pygame
import sys
from pygame.locals import *  # 将 pygame 的所有常量名导入

pygame.init()
size = width, height = 600, 400
bg = (255, 255, 255)
speed = [0, 0]
clock = pygame.time.Clock()
screen = pygame.display.set_mode(size)
pygame.display.set_caption("初次见面,请大家多多关照!")
turtle = pygame.image.load("turtle.png")
position = turtle.get_rect()
```

```
l_head = turtle # 龟头朝左
r_head = pygame.transform.flip(turtle, True, False) # 龟头朝左

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        if event.type == KEYDOWN:
            if event.key == K_LEFT:
                speed = [-1, 0]
                turtle = l_head
            if event.key == K_RIGHT:
                speed = [1, 0]
                turtle = r_head
            if event.key == K_UP:
                speed = [0, -1]
            if event.key == K_DOWN:
                speed = [0, 1]
    position = position.move(speed)
    if position.left < 0 or position.right > width:
        # 翻转图像
        turtle = pygame.transform.flip(turtle, True, False)
        # 反方向移动
        speed[0] = -speed[0]
    if position.top < 0 or position.bottom > height:
        speed[1] = -speed[1]
    screen.fill(bg)
    screen.blit(turtle, position)
    pygame.display.flip()
    clock.tick(30)
```

## 16.5 提高游戏的颜值



毋庸置疑，高颜值的界面会给你的游戏带来更多的眼球。

### 16.5.1 显示模式

前面通过 display 模块的 set_mode()方法来指定界面的大小，并返回一个 Surface 对象。set_mode()方法的原型如下：

```
set_mode(resolution=(0,0), flags=0, depth=0) -> Surface
```

第一个参数 resolution 用于指定界面的大小。一般会指定一个具体的尺寸，如果你什么都不给它，或者使用默认的(0，0)，那么 Pygame 会根据当前的屏幕分辨率创建一个窗口(SDL 版本低于 1.2.10 会抛出异常)。

第二个参数 flags 用于指定扩展选项。同时指定多个选项可以用管道操作符(|)隔开，表 16-2 列举了各个选项及其含义。

表 16-2 flags 可用的选项及含义

| 选　项 | 含　义 |
| --- | --- |
| FULLSCREEN | 全屏模式 |
| DOUBLEBUF | 双缓冲模式 |
| HWSURFACE | 硬件加速支持(只有在全屏模式下才能使用) |
| OPENGL | 使用 OpenGL 渲染 |
| RESIZABLE | 使得窗口可以调整大小 |
| NOFRAME | 使得窗口没有边框和控制按钮 |

第三个参数 depth 用于指定颜色位数。一般这个值不推荐设置,因为 Pygame 会自动根据当前操作系统设置最合适的颜色位数。

## 16.5.2 全屏才是王道

大家有没有发现:我们玩的很多游戏都是全屏模式,你知道为什么吗?因为全屏的好处太多了,例如可以显示更多的内容,可以开启硬件加速,最重要的一点是可以霸占着整个屏幕,其他的软件都一边站去。

开启全屏模式很简单,只需要设置第二个参数为 FULLSCREEN 即可,同时可以加上硬件加速 HWSURFACE:

```
screen = pygame.display.set_mode((640, 480), FULLSCREEN | HWSURFACE)
```

此时,你先别急着尝试运行代码,因为你这样毫无准备地使用全屏模式,稍后就很难退出了(如果你不幸已经进入了全屏模式,请用热键 Ctrl-Alt-Delete 退出)。所以,应该添加一个快捷键使得全屏模式得到控制:

```
# p16_3/pg_1.py
…
fullscreen = False
…
            # 全屏(F11)
            if event.key == K_F11:
                fullscreen = not fullscreen
                if fullscreen:
                    screen = pygame.display.set_mode((1920, 1080), \
                    FULLSCREEN | HWSURFACE)
                else:
                    screen = pygame.display.set_mode(size)
…
```

为了确保可以正常关闭程序,使用 F11 作为切换全屏模式到窗口模式的快捷键,这里已知显示器的当前分辨率是 1920×1080 像素,所以设置全屏后的尺寸为显示器的尺寸。但是你的游戏应该是给大家玩的,所以不同机器的显示器分辨率不可能完全相同,所以你需要获得当前显示器支持的分辨率。可以用 list_modes 方法实现:

```
>>> pygame.display.list_modes()
[(1920, 1080), (1680, 1050), (1600, 900), (1440, 900), (1400, 1050), (1366, 768), (1360, 768),
(1280, 1024), (1280, 800), (1280, 768), (1280, 720), (1024, 768), (800, 600), (640, 480), (640,
```

```
400), (512, 384), (400, 300), (320, 240), (320, 200)]
```

如上所示，list_modes()返回一个列表，从大到小依次列举出当前显示器支持的全屏分辨率。

### 16.5.3 使窗口尺寸可变

Pygame 的窗口默认是不可通过拖动边框来修改尺寸的，因为游戏角色、场景都是按照一定的比例来设计的，给你这么一拖一拽，男主一号都变成面目狰狞的坏蛋了！尽管如此，通过设置 RESIZABLE 选项还是可以实现的：

```
#这里由于空间有限,省略了大部分代码,完整代码可查阅源文件
# p16_3/pg_2.py
…
screen = pygame.display.set_mode(size, RESIZABLE)
…
            # 用户调整窗口尺寸
            if event.type == VIDEORESIZE:
                size = event.size
                width, height = size
                print(size)
                screen = pygame.display.set_mode(size, RESIZABLE)
…
```

开启了窗口尺寸可修改选项后，一旦用户调整窗口的尺寸，Pygame 就会发送一条带有最新尺寸的 VIDEORESIZE 事件到事件序列中。程序随即做出响应，重新设置 width 和 height 的值并重建一个新尺寸的窗口。

### 16.5.4 图像的变换

想要让程序实现更加炫酷的特技效果，你的图像还需要能够支持一些变换才行。例如左右上下翻转，按角度转动，放大缩小……Pygame 的 transform 模块使得你可以对图像（也就是 Surface 对象）做各种变换动作，并返回变换后的 Surface 对象。表 16-3 列举了 transform 模块几个常用的方法及作用。

**表 16-3 transform 模块的常用方法及作用**

| 方法 | 作用 | 方法 | 作用 |
|---|---|---|---|
| flip | 上下、左右翻转图像 | scale2x | 快速放大一倍图像 |
| scale | 缩放图像（快速） | smoothscale | 平滑缩放图像（精准） |
| rotate | 旋转图像 | chop | 裁剪图像 |
| rotozoom | 缩放并旋转图像 | | |

其实 transform 模块的这些方法都是像素转换的把戏，原理是通过使用一定的算法对图片进行像素位置修改。大多数方法在变换后难免会有一些精度的损失（flip()方法不会），因此不建议对变换后的 Surface 对象进行再次变换。

在前面小乌龟的例子中，就是采用 flip()方法让小乌龟可以在撞墙后自动“掉头”。接下来修改代码，实现小乌龟的缩放：

```
# p16_3/pg_3.py
…
ratio = 1.0 # 设置放大缩小比率
oturtle = pygame.image.load("turtle.png")
turtle = oturtle
oturtle_rect = oturtle.get_rect()
position = turtle_rect = oturtle_rect
…
# 放大、缩小小乌龟(=、-),空格恢复原始尺寸
if event.key == K_EQUALS or event.key == K_MINUS or event.key == K_SPACE:
    # 最大只能放大一倍,缩小 50%
    if event.key == K_EQUALS and ratio < 2:
        ratio += 0.1
    if event.key == K_MINUS and ratio > 0.5:
        ratio -= 0.1
    if event.key == K_SPACE:
        ratio = 1
    turtle = pygame.transform.smoothscale(oturtle, (int(oturtle_rect.width * ratio), int
(oturtle_rect.height * ratio)))
    # 相应修改龟头两个朝向的 Surface 对象,否则一单击移动就打回原形
    l_head = turtle
    r_head = pygame.transform.flip(turtle, True, False)
    # 获得小乌龟缩放后的新尺寸
    turtle_rect = turtle.get_rect()
position.width, position.height = turtle_rect.width, turtle_rect.height
…
```

接下来通过 rotate()方法让小乌龟实现贴边行走。

先来分析以下：rotate(Surface, angle)方法的第二个参数 angle 指定旋转的角度,是逆时针方向旋转的。而我们的小乌龟是这样,如图 16-7 所示。



图 16-7 小乌龟

每次 90 度的逆时针旋转结果如图 16-8 所示。

因此,代码这么写：

```
# p16_3/pg_4.py
import pygame
```

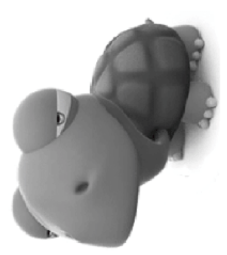
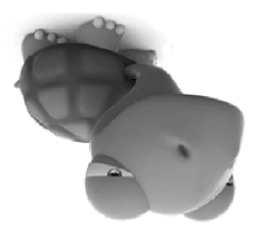





图 16-8 逆时针旋转的小乌龟

```
import sys
from pygame.locals import *

pygame.init()
```

```
size = width, height = 640, 480
bg = (255, 255, 255)

clock = pygame.time.Clock()
screen = pygame.display.set_mode(size)
pygame.display.set_caption("FishC Demo")
turtle = pygame.image.load("turtle.png")
position = turtle_rect = turtle.get_rect()
# 小乌龟顺时针行走
speed = [5, 0]
turtle_right = pygame.transform.rotate(turtle, 90)
turtle_top = pygame.transform.rotate(turtle, 180)
turtle_left = pygame.transform.rotate(turtle, 270)
turtle_bottom = turtle
# 刚开始走顶部
turtle = turtle_top

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
    position = position.move(speed)
    if position.right > width:
        turtle = turtle_right
        # 变换后矩形的尺寸发生改变
        position = turtle_rect = turtle.get_rect()
        # 矩形尺寸的改变导致位置也有变化
        position.left = width - turtle_rect.width
        speed = [0, 5]
    if position.bottom > height:
        turtle = turtle_bottom
        position = turtle_rect = turtle.get_rect()
        position.left = width - turtle_rect.width
        position.top = height - turtle_rect.height
        speed = [-5, 0]
    if position.left < 0:
        turtle = turtle_left
        position = turtle_rect = turtle.get_rect()
        position.top = height - turtle_rect.height
        speed = [0, -5]
    if position.top < 0:
        turtle = turtle_top
        position = turtle_rect = turtle.get_rect()
        speed = [5, 0]
    screen.fill(bg)
    screen.blit(turtle, position)
    pygame.display.flip()
    clock.tick(30)
```

### 16.5.5 裁剪图像

有些读者此前可能尝试使用 chop()方法写一个裁剪工具，但结果却事与愿

违。这是为啥呢？尝试在小乌龟的中间裁剪掉 50×50 像素后，看看是什么样子？

大家看下前后对比图，调用 chop()方法前小乌龟眉清目秀、气宇轩昂，如图 16-9 所示。

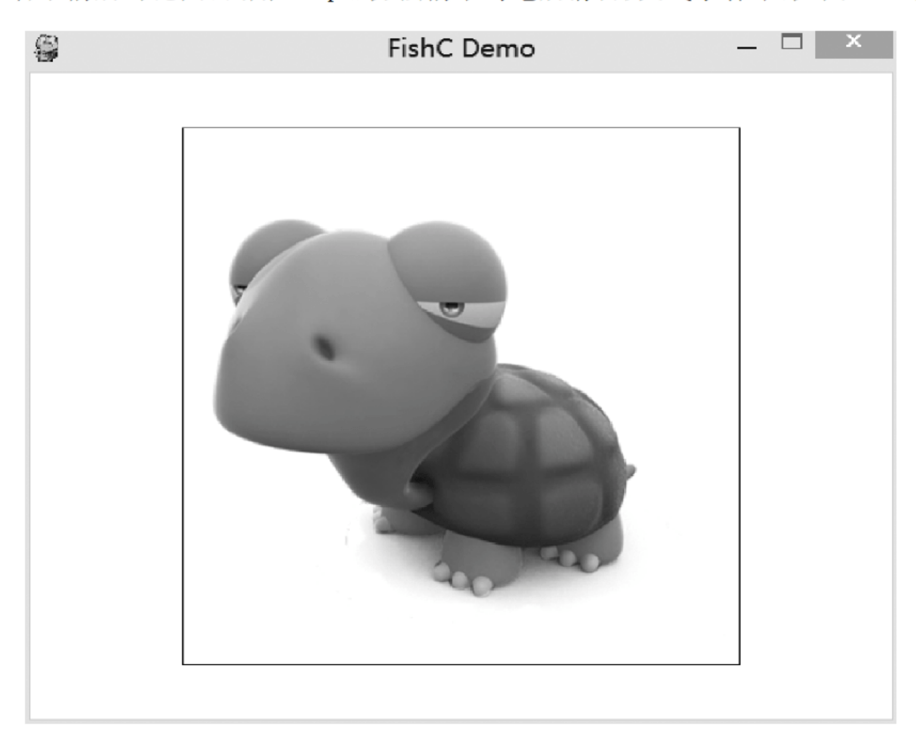



图 16-9　调用 chop()方法前

调用 chop()后小乌龟面目全非，惨不忍睹，如图 16-10 所示。

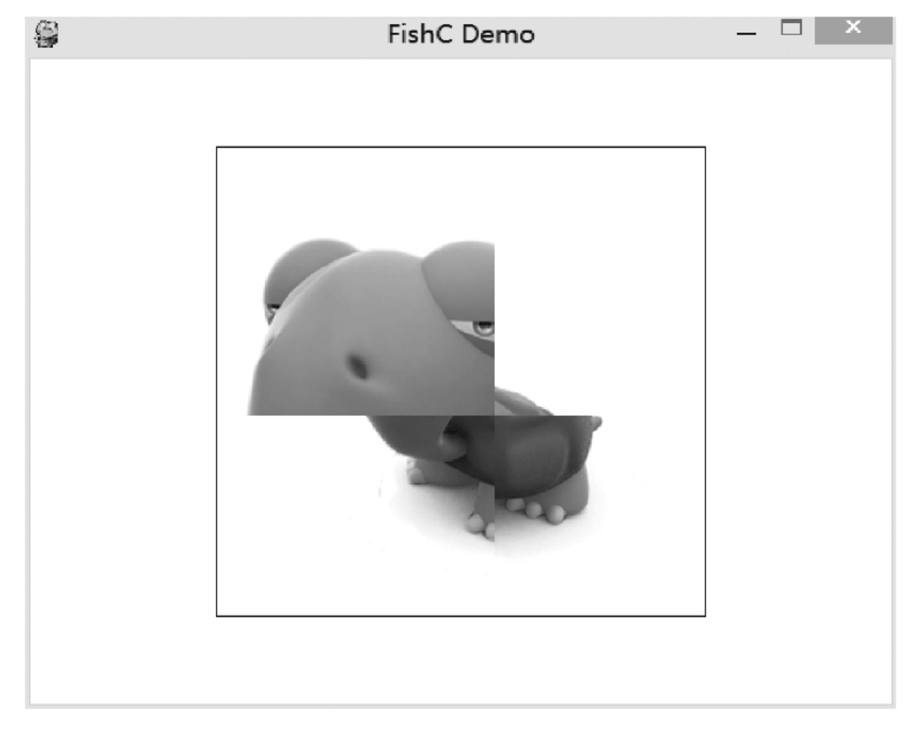



图 16-10　调用 chop()方法后

从对比中也不难看出这个 chop()方法是将指定的 Rect 矩形部分直接去掉，然后其他部分拼凑在一起返回 Surface 对象。

那要实现真正意义上的裁剪应该如何做呢？这个目前对我们来说有点小难度——难点就是鼠标每次按下到释放均有不同的意义。

先来分析，第一次拖动鼠标左键确定裁剪的范围，如图 16-11 所示。




图 16-11　第一次拖动鼠标确定裁剪的范围

第二次拖动鼠标左键裁剪范围内的图像，如图 16-12 所示。

第三次单击则表示重新开始，如图 16-13 所示。

这里用 draw.rect()来绘制矩形：

```
rect(Surface, color, Rect, width = 0)  -> Rect
```

- 第一个参数指定矩形将绘制在哪个 Surface 对象上；
- 第二个参数指定颜色；
- 第三个参数指定矩形的范围(left，top，width，height)；
- 第四个参数指定矩形边框的大小(0 表示填充矩形)。

裁剪操作可以利用 subsurface()方法来获得指定位置的子图像，然后 copy()出来：

```
capture = screen.subsurface(select_rect).copy()
```

正如刚才所提到的，这个例子的难度主要在于区分每次单击的操作，因此不妨使用两个变量来做标志：

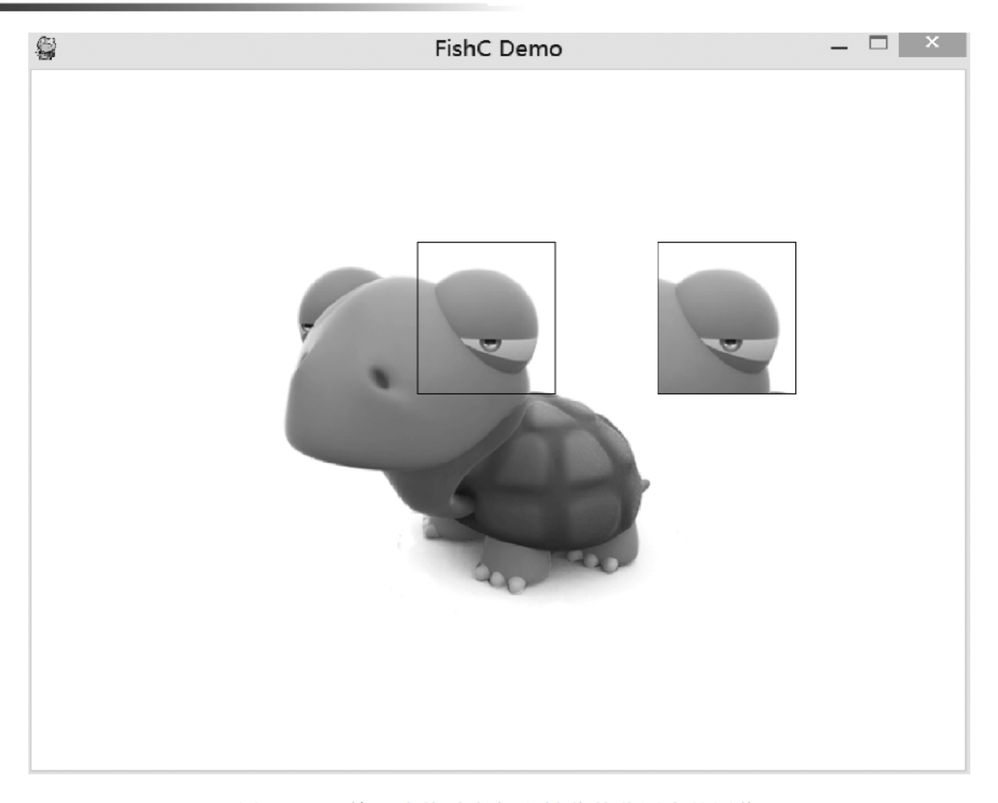



图 16-12　第二次拖动鼠标左键裁剪范围内的图像

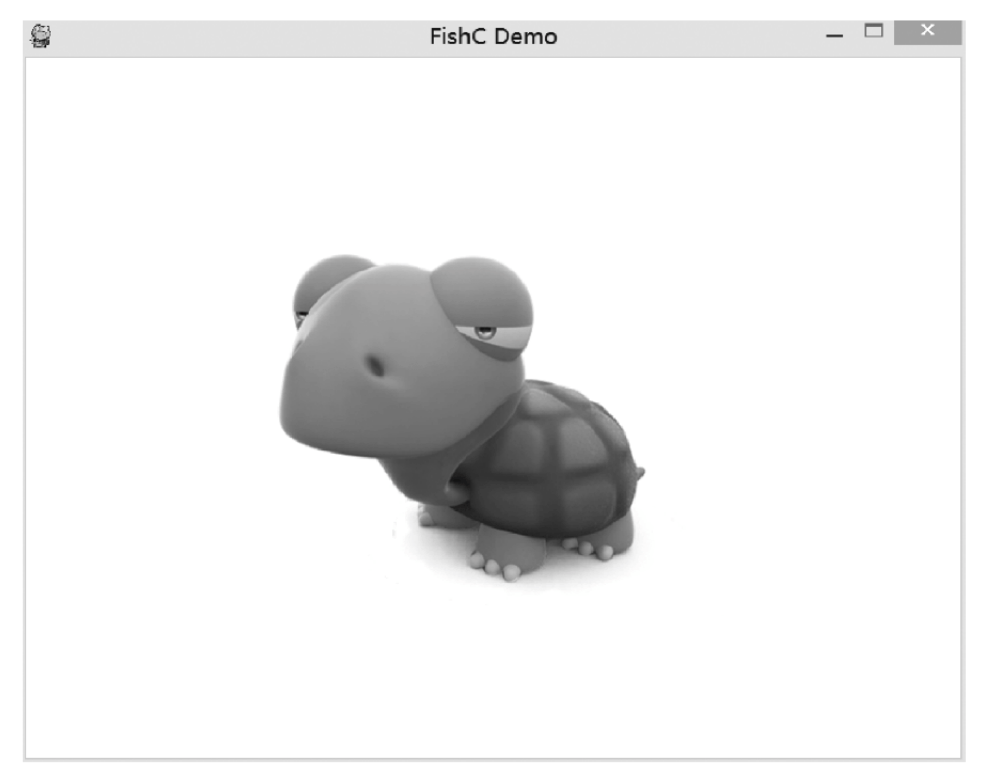



图 16-13　第三次单击则表示重新开始

```
# 0 -> 未选择,1 -> 选择中,2 -> 完成选择
select = 0
# 0 -> 未拖动,1 -> 拖动中,2 -> 完成拖动
drag = 0
…
if event.type == MOUSEBUTTONDOWN:
    if event.button == 1:
        # 第一次单击,选择范围
        if select == 0 and drag == 0:
            …
            select = 1
        # 第二次单击,推拽图像
        elif select == 2 and drag == 0:
            …
            drag = 1
        # 第三次单击,初始化
        elif select == 2 and drag == 2:
            select = 0
            drag = 0
if event.type == MOUSEBUTTONUP:
    if event.button == 1:
        # 第一次释放,结束选择
        if select == 1 and drag == 0:
            …
            select = 2
        # 第二次释放,结束拖动
        if select == 2 and drag == 1:
            drag = 2
screen.fill(bg)
screen.blit(turtle, position)
# 实时绘制选择框
if select:
    # mouse.get_pos() 用于获取鼠标当前位置
    mouse_pos = pygame.mouse.get_pos()
…
# 拖动裁剪的图像
if drag:
    …
```

完整代码参考：

```
# p16_4/Demo.py
import pygame
import sys
from pygame.locals import *

pygame.init()
size = width, height = 800, 600
bg = (255, 255, 255)
clock = pygame.time.Clock()
screen = pygame.display.set_mode(size)
pygame.display.set_caption("FishC Demo")
```

```
turtle = pygame.image.load("turtle.png")
# 0 -> 未选择,1 -> 选择中,2 -> 完成选择
select = 0
select_rect = pygame.Rect(0, 0, 0, 0)
# 0 -> 未拖动,1 -> 拖动中,2 -> 完成拖动
drag = 0
position = turtle.get_rect()
position.center = width // 2, height // 2

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        elif event.type == MOUSEBUTTONDOWN:
            if event.button == 1:
                # 第一次单击,选择范围
                if select == 0 and drag == 0:
                    pos_start = event.pos
                    select = 1
                # 第二次单击,推拽图像
                elif select == 2 and drag == 0:
                    capture = screen.subsurface(select_rect).copy()
                    cap_rect = capture.get_rect()
                    drag = 1
                # 第三次单击,初始化
                elif select == 2 and drag == 2:
                    select = 0
                    drag = 0
        elif event.type == MOUSEBUTTONUP:
            if event.button == 1:
                # 第一次释放,结束选择
                if select == 1 and drag == 0:
                    pos_stop = event.pos
                    select = 2
                # 第二次释放,结束拖动
                if select == 2 and drag == 1:
                    drag = 2
    screen.fill(bg)
    screen.blit(turtle, position)
    # 实时绘制选择框
    if select:
        mouse_pos = pygame.mouse.get_pos()
        if select == 1:
            pos_stop = mouse_pos
        select_rect.left, select_rect.top = pos_start
        select_rect.width, select_rect.height = pos_stop[0] - pos_start[0], pos_stop[1] -
pos_start[1]
        pygame.draw.rect(screen, (0, 0, 0), select_rect,1)
    # 拖动裁剪的图像
    if drag:
        if drag == 1:
            cap_rect.center = mouse_pos
```

```
        screen.blit(capture, cap_rect)
    pygame.display.flip()
clock.tick(30)
```

### 16.5.6 转换图片

图像是特定像素的组合，而 Surface 对象是 Pygame 对图像的描述。在 Pygame 中，到处都是 Surface 对象：set_mode()方法返回的是一个 Surface 对象；在界面上打印文字，也是先将文字转变成 Surface 对象再"贴"上去；小乌龟在上边爬来爬去，事实上就是不断调整 Surface 对象上一些特定像素的位置。

image.load()载入图片后将返回一个 Surface 对象，此前我们一直拿来就用，没有对其进行转换，这是效率相对较低的做法。如果你希望 Pygame 尽可能高效地处理图片，那么应该在载入图片后同时调用 convert()方法进行转换：

```
background = pygame.image.load("background.jpg").convert()
```

有读者可能会好奇：不是说 image.load()会返回一个 Surface 对象吗？还转换个啥？

其实这里转换的是"像素格式"，image.load()返回的 Surface 对象中保留了原图像的像素格式。在调用 blit()方法的时候，如果两个 Surface 对象的像素格式不同，那么 Pygame 会实时地进行转换，这是相当费时的操作。

还有一个是 convert_alpha()，它们有什么区别呢？一般情况下用 RGB 来描述一个颜色，而在游戏开发中常常用 RGBA 来描述。多的这个 A 指的是 Alpha 通道，用于表示透明度，它的值也是 0～255，0 表示完全透明，255 表示完全不透明。image.load()支持多种格式的图片导入，对于包含 alpha 通道的图片，使用 convert_alpha()转换格式，否则使用 convert()：

```
turtle = pygame.image.load("turtle.png").convert_alpha()
```

### 16.5.7 透明度分析

Pygame 支持三种类型的透明度设置：colorkeys、surface alphas 和 pixel alphas。设置 colorkeys 是指定一种颜色，使其变为透明。surface alphas 是整体设置一个图片的透明度。pixel alphas 为每个像素增加一个 alpha 通道，也就是允许设置每个像素的透明度。colorkeys 和 surface alphas 可以混合使用，而 pixel alphas 不能和其他类型混合。

说得那么复杂，其实就是由 convert()方法转换来的 Surface 对象支持 colorkeys 和 surface alphas 设置透明度，并且可以混合设置。而 convert_alpha()方法转换后是支持 pixel alphas，也就是这个图片本身每个像素都带有 alpha 通道(所以载入一个带 alpha 通道的 png 图片，可以看到该图片部分位置是透明的)。

接下来做个实验：这里有两张图片 turtle.jpg 和 turtle.png。turtle.jpg 不带 alpha 通道，turtle.png 带 alpha 通道，并且背景被设置为透明。

首先载入 turtle.jpg，使用 set_colorkey()方法试图将白色的背景透明化：

```
# p16_5/pg_1.py
…
turtle.set_colorkey((255, 255, 255))
turtle.set_alpha(200)
…
```

程序实现结果并不是很优秀，如图 16-14 所示。

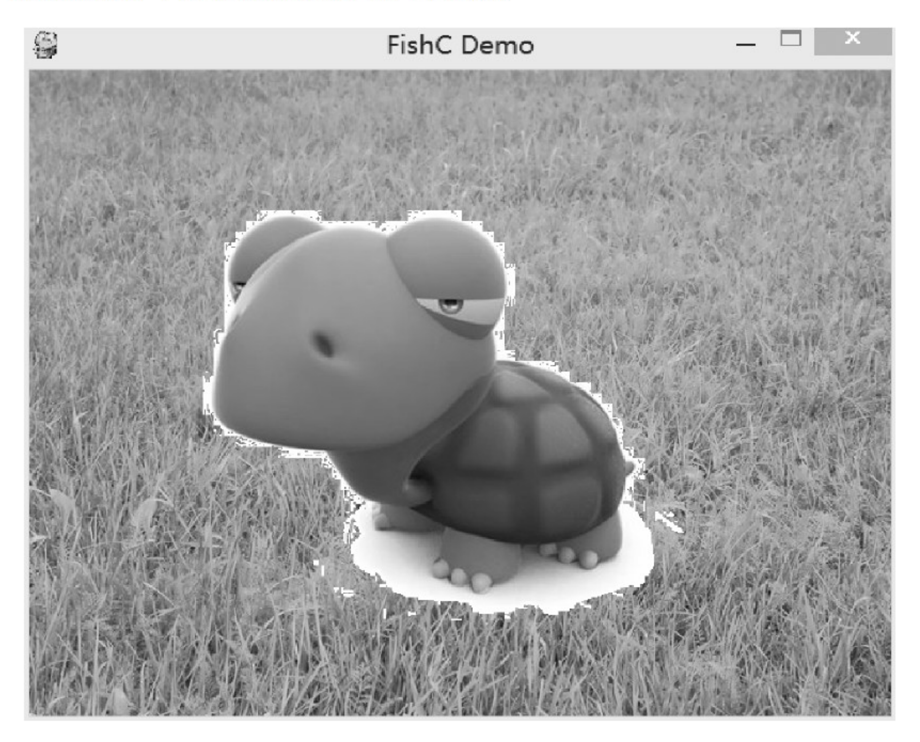



图 16-14　使用 set_colorkey()方法试图将白色的背景透明化

使用 set_alpha()方法将调节整个图片的透明度，如图 16-15 所示。




图 16-15　使用 set_alpha()方法将调节整个图片的透明度

另外，set_colorkey()和 set_alpha()是可以混合使用的，如图 16-16 所示。

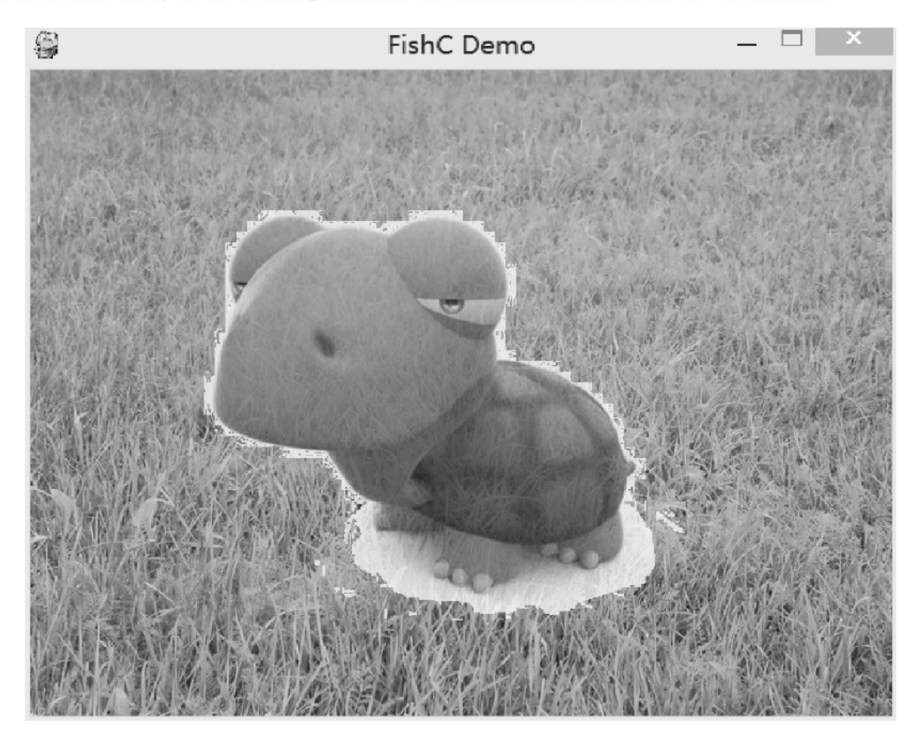



图 16-16　将 set_colorkey()方法和 set_alpha()方法混合使用

最后是 pixel alphas，turtle.png 这个图片是带有 alpha 通道的，并且背景被设置为透明，因此直接载入后可以看到透明的背景，如图 16-17 所示。




图 16-17　turtle.png 这个图片是带有 alpha 通道的，并且背景被设置为透明

但如果希望调节小乌龟自身的透明度，可以用 get_at()获取单个像素的颜色，并用 set_at()来修改它。get_at()和 set_at()使用的是 RGBA 颜色，也就是带 Alpha 通道的 RGB 颜色：

```
print(turtle.get_at(position.center))
```

因此，如果想将整个小乌龟的透明度调整为 200 的时候，可以逐个像素修改透明度：

```
# p16_5/pg_2.py
…
for i in range(position.width):
    for j in range(position.height):
        temp = turtle.get_at((i, j))
        if temp[3] != 0:
            temp[3] = 200
        turtle.set_at((i, j), temp)
…
```

效果竟然还是不理想，如图 16-18 所示。




图 16-18　通过逐个设置像素值将整个小乌龟的透明度调整为 200

没关系，程序是死的，程序员是活的！这里教大家一个新技能来 Get 这个问题。先给大家看解决方案，再分析：

```
# p16_5/pg_3.py
…
def blit_alpha(target, source, location, opacity):
    x = location[0]
    y = location[1]
    temp = pygame.Surface((source.get_width(), source.get_height())).convert()
    temp.blit(target, (-x, -y))
```

```
    temp.blit(source, (0, 0))
    temp.set_alpha(opacity)
    target.blit(temp, location)
…
    blit_alpha(screen, turtle, position, 200)
…
```

程序实现效果如图 16-19 所示。




图 16-19 调整图片透明度的新技能

嗯，这是我们想要的结果！那来看看这个函数是如何做到的：

（1）首先创造一个不带 alpha 通道的小乌龟；

（2）然后将小乌龟所在位置的背景覆盖上去；

（3）此刻 temp 得到的是一个跟小乌龟尺寸一样大小，上边绘制着背景的 Surface 对象；

（4）将带 alpha 通道的小乌龟覆盖上去；

（5）由于 temp 是不带 alpha 通道的 Surface 对象，因此使用 set_alpha()方法设置整个图片的透明度；

（6）最后将设置好透明度的 temp“贴”到指定位置上，完成任务！

## 16.6 绘制基本图形



有些读者可能会说，前边你不是才说大部分的游戏都是由图片构成的吗？不是说颜值对于一个游戏来说有多重要吗？我们学 Pygame 就是为了游戏开发，那绘制基本的图形对于游戏开发有什么用？作者你这是在自己打脸吗？

其实，绘制基本图形在游戏开发中并不是没用！说来也奇怪，最近很火的游戏反而是一些

像素游戏，尤其是一些由简单图形构成的小游戏。总结了一下有几点原因：

(1) 唯美的游戏界面越来越多，玩家难免出现审美疲劳；

(2) 时下盛行极简风格，只要你的游戏做得让玩家舒服，一般大家都不会拒绝简单的游戏；

(3) 大型的游戏 CG 动画绘制需要耗费相当的人力、物力和财力；

(4) 简单的游戏更容易开发，小游戏工作室或个人即可完成开发，有更多逆袭的机会；

(5) 游戏依托的主要平台已经从电脑端转移到手机端，那么小一个屏幕你把图像做得惟妙惟肖意义并不大。其实衍生出来还有很多原因，例如手机的配置差异大，而游戏需要尽可能满足配置低的手机才能获得更多的玩家。大型游戏消耗大，耗电、散热都是一个需要考虑的问题。另外，简单的图形也能构造出高颜值的游戏，越是简单越是抽象，越是抽象越是艺术嘛。

Pygame 的 draw 模块提供了绘制简单图形的方法，支持绘制的图形有矩形、多边形、圆形、椭圆形、弧形和线条。

### 16.6.1 绘制矩形

绘制矩形的语句格式如下：

```
rect(Surface, color, Rect, width = 0)
```

- 第一个参数指定矩形将绘制在哪个 Surface 对象上；
- 第二个参数指定颜色；
- 第三个参数指定矩形的范围(left, top, width, height)；
- 第四个参数指定矩形边框的大小(0 表示填充矩形)。

rect()方法用于在 Surface 对象上边绘制一个矩形，关于最后一个参数 width，看下面的例子：

```
# p16_6/pg_1.py
...
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
...
    pygame.draw.rect(screen, BLACK, (50, 50, 150, 50), 0)
    pygame.draw.rect(screen, BLACK, (250, 50, 150, 50), 1)
    pygame.draw.rect(screen, BLACK, (450, 50, 150, 50), 10)
...
    # 限制每秒绘制 10 次，否则 CPU 会跑满
    clock.tick(10)
```

程序实现如图 16-20 所示，width 为 0 表示填充整个矩形，边框是向外延伸的。

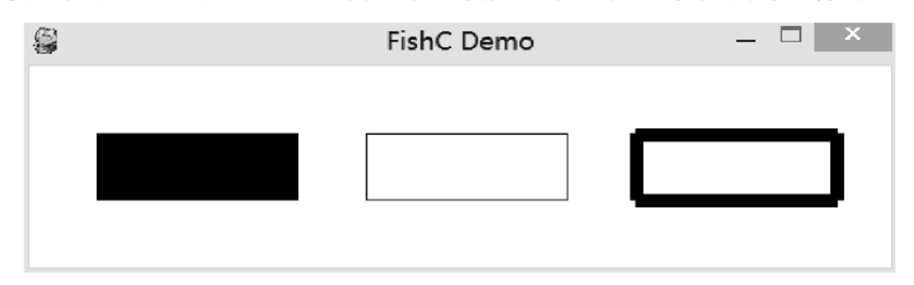



图 16-20　绘制矩形

### 16.6.2 绘制多边形

绘制多边形的语句格式如下：

```
polygon(Surface, color, pointlist, width=0)
```

polygon()的用法跟 rect()类似，除了第三个参数不同，polygon()方法的第三个参数接受由多边形各个顶点坐标组成的列表。

```
# p16_6/pg_2.py
…
points = [(200, 75), (300, 25), (400, 75), (450, 25), (450, 125), (400, 75), (300, 125)]
…
pygame.draw.polygon(screen, GREEN, points, 0)
…
```

程序实现如图 16-21 所示。

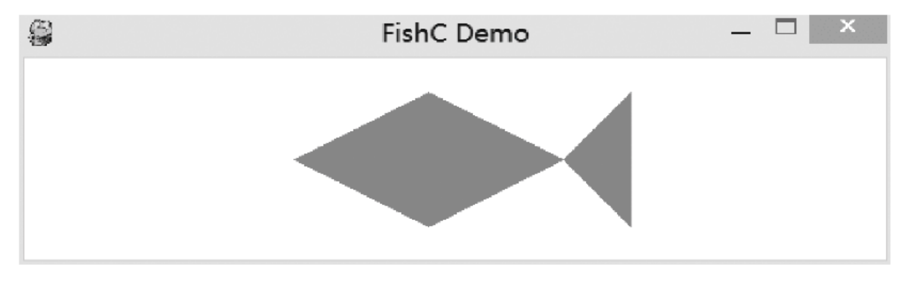



图 16-21 绘制多边形

### 16.6.3 绘制圆形

绘制圆形的语句格式如下：

```
circle(Surface, color, pos, radius, width=0)
```

第一、二、五个参数跟前面的两个方法一样，第三个参数指定圆心的位置，第四个参数指定半径的大小。看下面的例子：

```
# p16_6/pg_3.py
…
position = size[0]//2, size[1]//2
moving = False
…
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                moving = True
        if event.type == pygame.MOUSEBUTTONUP:
            if event.button == 1:
```

```
                moving = False
    if moving:
        position = pygame.mouse.get_pos()
    screen.fill(WHITE)
    pygame.draw.circle(screen, RED, position, 25, 1)
    pygame.draw.circle(screen, GREEN, position, 75, 1)
    pygame.draw.circle(screen, BLUE, position, 125, 1)
…
```

程序实现如图 16-22 所示。

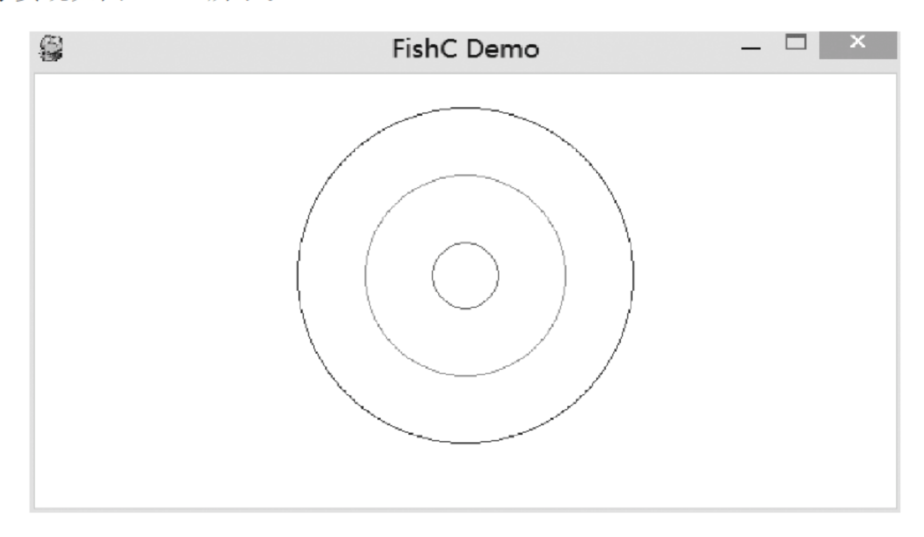



图 16-22　绘制圆形

### 16.6.4　绘制椭圆形

绘制椭圆形的语句格式如下：

```
ellipse(Surface, color, Rect, width = 0)
```

椭圆是利用第三个参数指定的矩形来绘制的，所以你的限定矩形如果是正方形，那么画出来的就是一个圆形了。

```
# p16_6/pg_4.py
…
    pygame.draw.ellipse(screen, BLACK, (100, 100, 440, 100), 1)
    pygame.draw.ellipse(screen, BLACK, (220, 50, 200, 200), 1)
…
```

程序实现如图 16-23 所示。

### 16.6.5　绘制弧线

绘制弧线的语句格式如下：

```
arc(Surface, color, Rect, start_angle, stop_angle, width = 1)
```

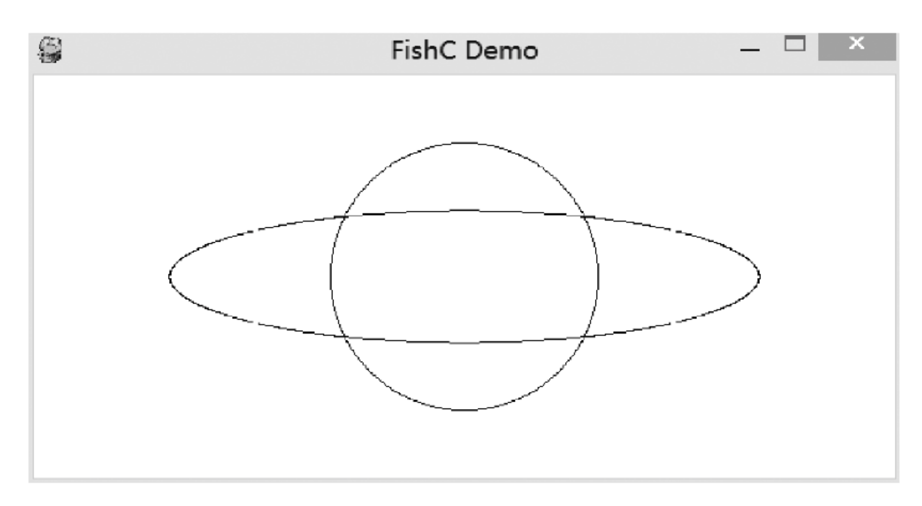



图 16-23　绘制椭圆形

arc()方法是绘制椭圆弧，也就绘制椭圆上的一部分弧线，因为弧线并不是全包围图形，所以不能将 width 设置为 0 进行填充。start_angle 和 stop_angle 参数用于设置弧线的起始角度和结束角度，单位是弧度。

```
# p16_6/pg_5.py
…
import math
…
    pygame.draw.arc(screen, BLACK, (100, 100, 440, 100), 0, math.pi, 1)
    pygame.draw.arc(screen, BLACK, (220, 50, 200, 200), math.pi, 2 * math.pi, 1)
…
```

程序实现如图 16-24 所示。

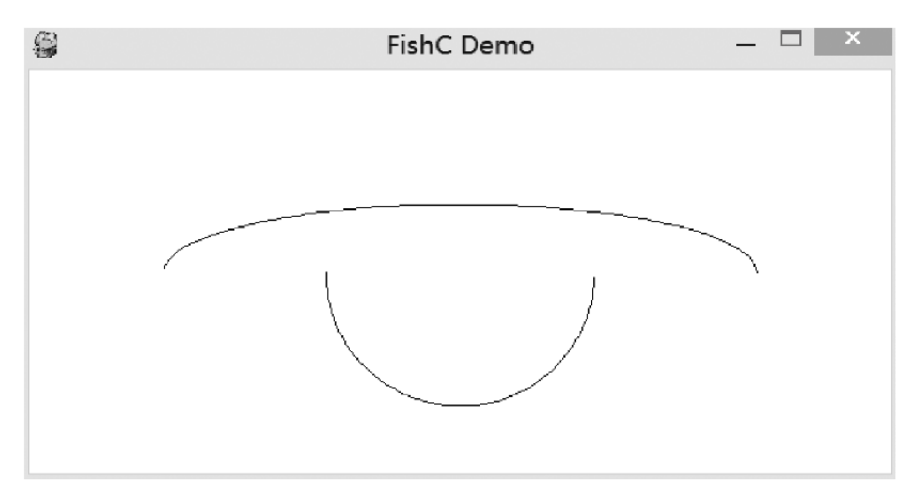



图 16-24　绘制弧线

## 16.6.6　绘制线段

绘制线段的语句格式如下：

```
line(Surface, color, start_pos, end_pos, width = 1)
lines(Surface, color, closed, pointlist, width = 1)
```

line()用于绘制一条线段，而 lines()则用于绘制多条线段。其中，lines()方法的 closed 参数设置是否首尾相连，跟 polygon()有点像，但区别是线段不能通过设置 width 参数为 0 进行填充。

```
aaline(Surface, color, startpos, endpos, blend = 1)
aalines(Surface, color, closed, pointlist, blend = 1)
```

经常玩游戏的读者应该听说过“抗锯齿”，开启抗锯齿后画面质量会有质的飞跃。没错，aaline()和 aalines()方法是用来绘制抗锯齿的线段，aa 就是 antialiased，抗锯齿的意思。最后一个参数 blend 指定是否通过绘制混合背景的阴影来实现抗锯齿功能。由于没有 width 方法，所以它们只能绘制 1 个像素的线段。

```
# p16_6/pg_6.py
…
    pygame.draw.lines(screen, GREEN, 1, points, 1)
    pygame.draw.line(screen, BLACK, (100, 200), (540, 250), 1)
    pygame.draw.aaline(screen, BLACK, (100, 250), (540, 300), 1)
pygame.draw.aaline(screen, BLACK, (100, 300), (540, 350), 0)
…
```

程序实现如图 16-25 所示。

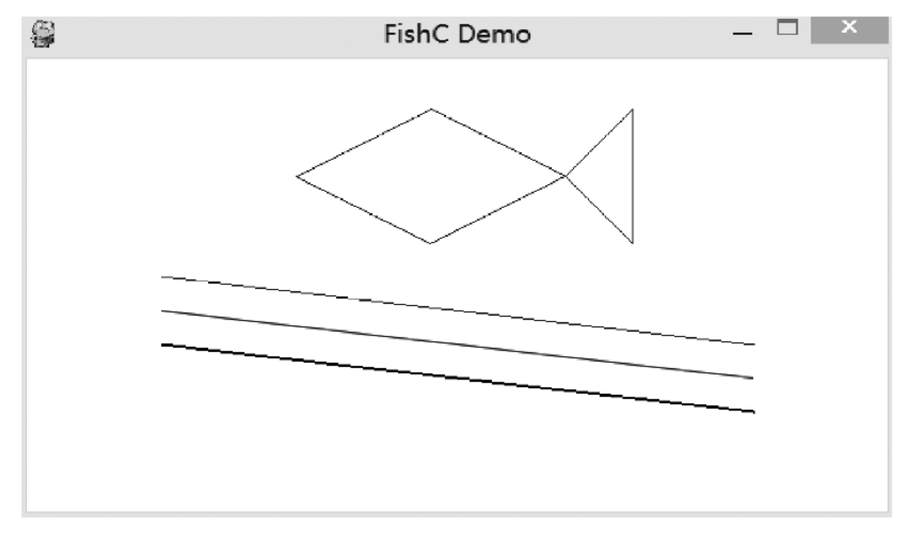



图 16-25　绘制线段

## 16.7　动画精灵



截至目前，我们已经学了 Pygame 的事件、图片的转换及移动、基本的图形绘制、透明度调整等内容，但距离真正实现一个游戏还差一个环节：碰撞检测。在讲碰撞检测之前需要引入一个新的知识：动画精灵。




Nonono，不是说蓝精灵。我们说的动画精灵是指游戏开发中，那些赋予灵魂的事物，像前边的小乌龟。动画精灵的实现看似简单，实际不然。因为在真正的游戏开发中，远远不只有一个精灵，它们的数量随时都会发生变化（比如说敌人不断地出现，然后不断地被消灭），它们的移动轨迹也并不是一样的，既然轨迹不同，那么肯定就会发生碰撞，所以精灵还要支持碰撞检测才行。

下边将通过一个小游戏的讲解来学习新的知识，同时体验一个游戏开发的过程。这个游戏我取名叫 PlayTheBall，中文名大概叫“玩个球啊”。代码量在两百行左右，但其中涉及碰撞检测、异常处理、计时器、自定义事件、播放声音、替换鼠标样式、限定鼠标移动范围等新的知识点。

游戏界面如图 16-26 所示。

### 1. 游戏介绍

游戏的背景是在不久的将来，人类过度开荒，地球资源不断枯竭……有一天，五大洲上出现了五个巨大的黑洞正在吞噬地球，地球危在旦夕……传闻只要集齐游荡于世界各地的金木水火土五颗神球，并分别将其置入黑洞中，就可以拯救地球。但由于环境污染严重，五个神球已经黯然无光……所以，我们需要做的，就是先摩擦摩擦！

### 2. 游戏说明

(1) 游戏伴随着魔性的音乐（《我的滑板鞋》）进行，界面上出现五个随机速度的灰色小球，它们会在相互碰撞后改变原来的速度。

(2) 如果小球从页面的上方穿过，会从下方出现，同样，如果小球从左边进入会从右边




图 16-26　PlayTheBall 游戏界面

出来。

(3) 鼠标的活动范围被限定在下方的玻璃面板上，通过一定频率的不断移动鼠标，会使得相应的小球从灰色变成绿色并停止移动，此时你可以使用 w、s、a、d 按键分别上下左右移动小球。

(4) 当玩家将绿色的小球移动到背景中黑洞的上方，按下空格键会检查该小球的位置是否完全覆盖黑洞，如果是的话，小球将被固定在黑洞中，此后其他球将忽略它，直接从它上方飘过。

(5) 需要注意的是，如果玩家的小球变绿色了，在放入黑洞前要时刻提防着其他球的碰撞，因为一旦发生碰撞，绿色的小球就会马上脱离你的控制(变成灰色)，并重新获得随机的速度。

(6) 在歌曲播完之前，如果玩家能把所有的小球都成功地固定在每个黑洞中，游戏胜利。

### 16.7.1　创建精灵

Pygame 的 sprite 模块提供了一个动画精灵的基类，游戏中的小球就是通过继承它而创建出来的精灵。

```
# p16_7/main.py
import pygame
import sys
from pygame.locals import *
from random import *

class Ball(pygame.sprite.Sprite): # 球类继承自 Spirte 类
```

```
    def __init__(self, image, position, speed):
        pygame.sprite.Sprite.__init__(self) # 初始化动画精灵
        self.image = pygame.image.load(image).convert_alpha()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = position # 将小球放在指定位置
        self.speed = speed

def main():
    pygame.init()
    ball_image = "gray_ball.png"
    bg_image = "background.png"
    running = True
    bg_size = width, height = 1024, 681 # 根据背景图片指定游戏界面尺寸
    screen = pygame.display.set_mode(bg_size)
    pygame.display.set_caption("Play the ball - FishC Demo")
    background = pygame.image.load(bg_image).convert_alpha()
    balls = [] # 用来存放小球对象的列表

    # 创建五个小球
    for i in range(5):
        # 位置随机,速度随机
        position = randint(0, width-100), randint(0, height-100)
        speed = [randint(-10, 10), randint(-10, 10)]
        ball = Ball(ball_image, position, speed)
        balls.append(ball)

    clock = pygame.time.Clock()

    while running:
        for event in pygame.event.get():
            if event.type == QUIT:
                sys.exit()
        screen.blit(background, (0, 0))
        for each in balls:
            screen.blit(each.image, each.rect)
        pygame.display.flip()
        clock.tick(30)

if __name__ == "__main__":
    main()
```

程序实现如图 16-27 所示。

## 16.7.2 移动精灵

接下来让小球动起来，事实上就是在 Ball 类中添加 move()方法，然后在绘制每个小球前先调用一次 move()移动到新的位置。

```
...
    def move(self):
        self.rect = self.rect.move(self.speed)
...
```

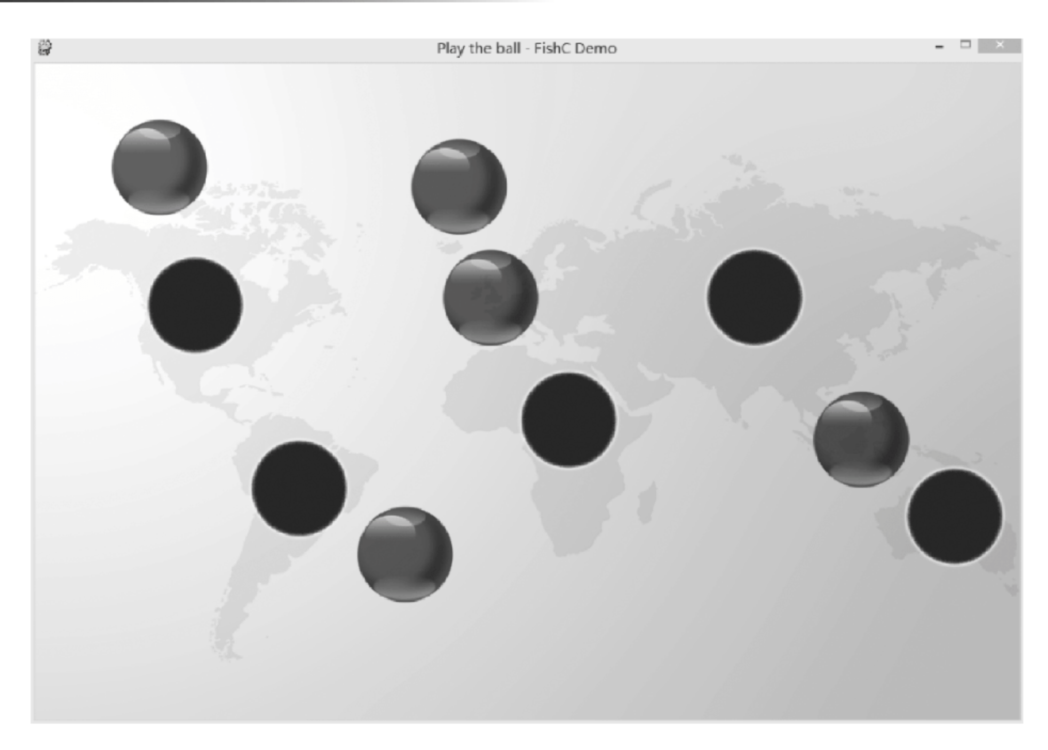



图 16-27　创建精灵

```
        for each in balls:
            each.move()
            screen.blit(each.image, each.rect)
...
```

如果小球从页面的上方穿过，会从下方出现，同样，如果小球从左边进入会从右边出来。

```
...
class Ball(pygame.sprite.Sprite):
    # 增加一个背景尺寸的参数
    def __init__(self, image, position, speed, bg_size):
        ...
        self.width, self.height = bg_size[0], bg_size[1]

    def move(self):
        self.rect = self.rect.move(self.speed)
        # 如果小球的右侧出了边界，那么将小球左侧的位置改为右侧的边界
        # 这样便实现了从左边进入，右边出来的效果
        if self.rect.right < 0:
            self.rect.left = self.width
        elif self.rect.left > self.width:
            self.rect.right = 0
        elif self.rect.bottom < 0:
            self.rect.top = self.height
        elif self.rect.top > self.height:
            self.rect.bottom = 0
...
```

这样小球就能在屏幕上自由穿越了，如图 16-28 所示。



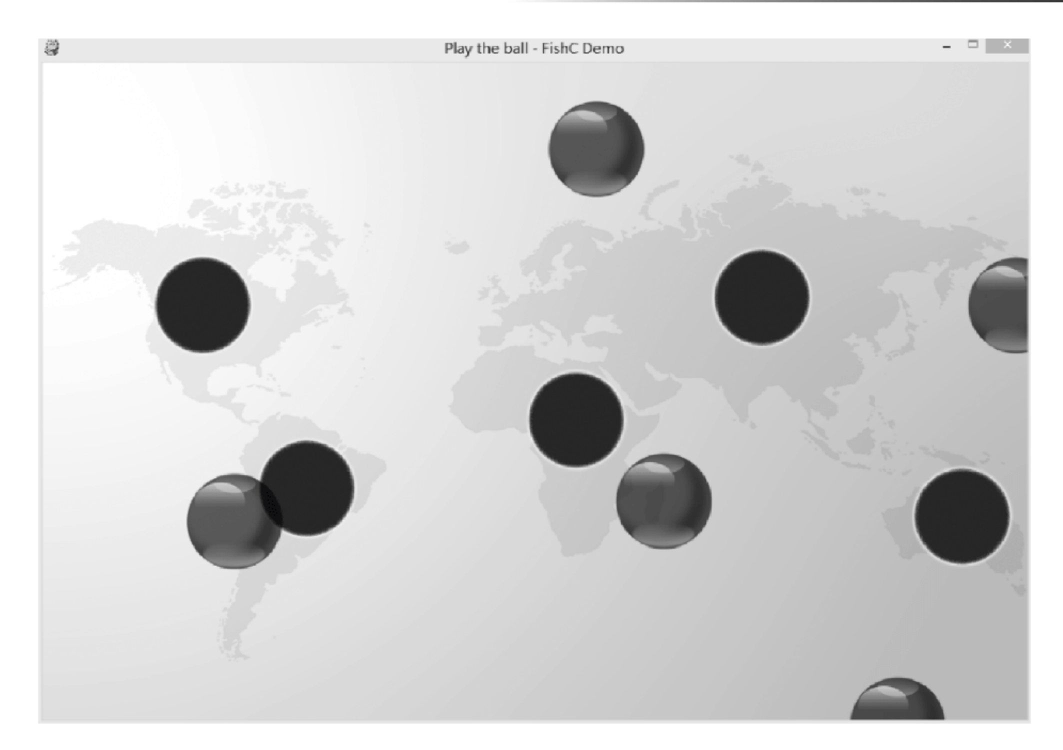



图 16-28　移动精灵

## 16.8　碰撞检测



大部分的游戏都需要做碰撞检测，比如需要知道小球是否发生了碰撞，子弹是否击中了目标，主角是否踩到了地雷。那应该如何实现呢？其实原理就是检查两个精灵之间是否存在重叠的部分。

### 16.8.1　尝试自己写碰撞检测函数

对于两个球来说，对比它们的圆心距离和半径的和即可，如图 16-29 到图 16-31 所示。

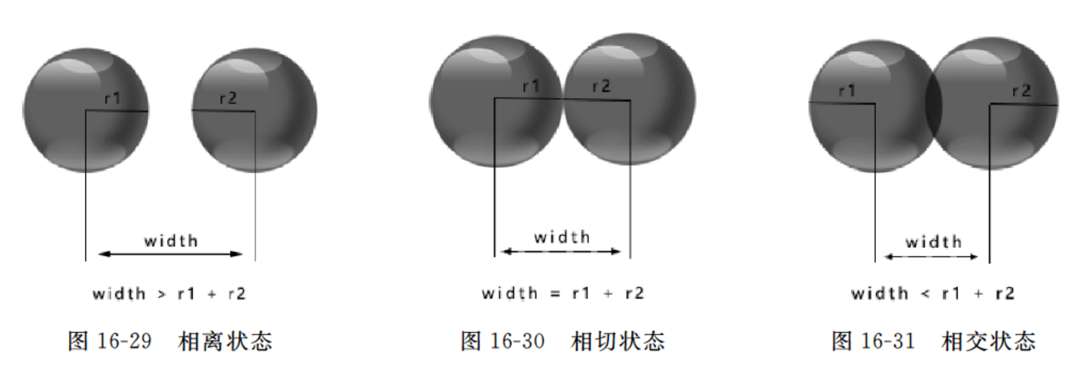



图 16-29　相离状态　　图 16-30　相切状态　　图 16-31　相交状态

下面是一个检测各个小球之间是否发生碰撞的函数，一旦发生便修改小球的移动方向：

```
# p16_8/collide_check.py
```

```
def collide_check(item, target):
    col_balls = []
    for each in target:
        distance = math.sqrt(\
            math.pow((item.rect.center[0] - each.rect.center[0]), 2) \
             + math.pow((item.rect.center[1] - each.rect.center[1]), 2))
        if distance <= (item.rect.width + each.rect.width) / 2:
            col_balls.append(each)
    return col_balls
…
        # 先让所有小球移动一步
        for each in balls:
            each.move()
            screen.blit(each.image, each.rect)
        # 检测各个小球之间是否发生碰撞
        for i in range(BALL_NUM):
            # 先将要检测的小球拿出来
            item = balls.pop(i)
            # 与列表中的其他小球一一对比
            if collide_check(item, balls):
                item.speed[0] = - item.speed[0]
                item.speed[1] = - item.speed[1]
            # 将小球放回到列表中
            balls.insert(i, item)
```

程序成功地实现了碰撞检测，但运气不大好的时候，会出现小球卡住的现象，如图 16-32 所示。

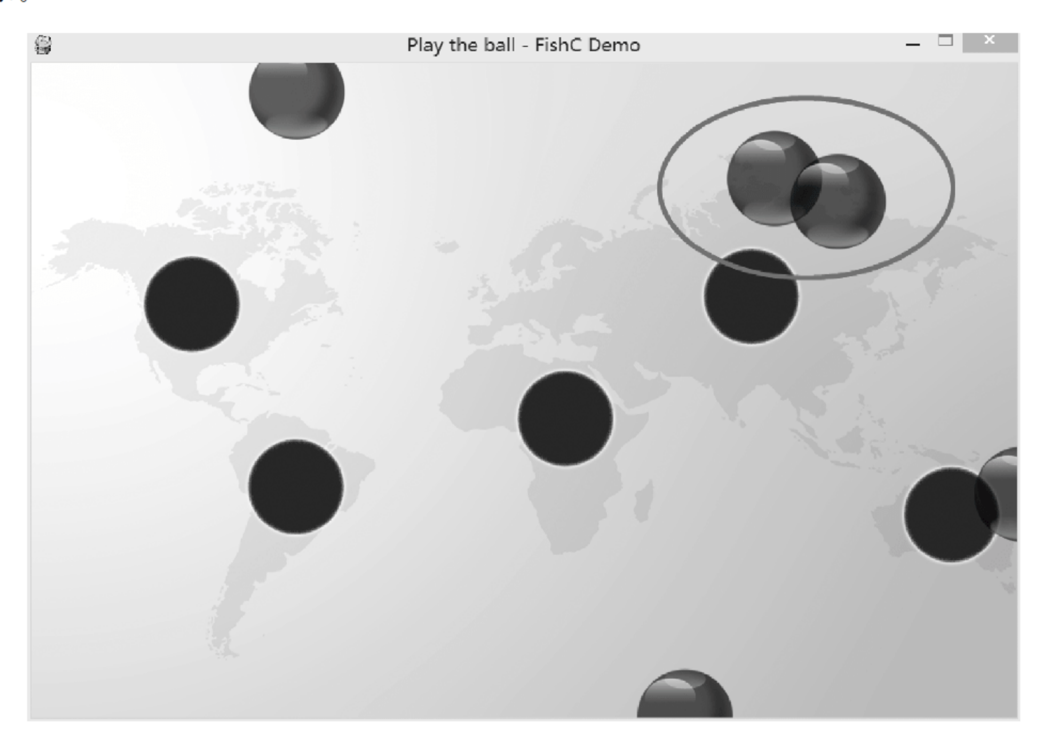



图 16-32　小球卡住了

原因是：当小球在诞生的位置恰好有其他小球，因此检测到两个小球发生碰撞，速度取反，但如果两个小球相互覆盖的范围大于移动一次的距离，那就会出现卡住的现象(反向移动后仍然检测到碰撞，则速度取反，又变成相向移动，速度不变的情况下是死循环)。

解决方案：在小球诞生的时候立刻检查该位置是否有其他小球，有的话修改新生小球的位置。

```
…
    # 创建五个小球
    BALL_NUM = 5
    for i in range(BALL_NUM):
        # 位置随机，速度随机
        position = randint(0, width - 100), randint(0, height - 100)
        speed = [randint(-10, 10), randint(-10, 10)]
        ball = Ball(ball_image, position, speed, bg_size)
        # 测试诞生小球的位置是否存在其他小球
        while collide_check(ball, balls):
            ball.rect.left, ball.rect.top = randint(0, width - 100), \
            randint(0, height - 100)
        balls.append(ball)
…
```

不过这个 collide_check()函数只适用于圆与圆间的碰撞检测，如果是其他多边形或不规则图形，那么就得不到相应的效果了。当然，对于聪明的读者朋友来说，为每一种特殊情况写一个检测函数也并不是不可以。Pygame 的 sprite 模块事实上已经提供了碰撞检测的函数供大家使用，这也正是为什么我们的类要继承自 sprite 模块的 Sprite 基类的原因。

## 16.8.2 sprite 模块提供的碰撞检测函数

sprite 模块提供了一个 spritecollide()函数，用于检测某个精灵是否与指定组中的其他精灵发生碰撞。

```
spritecollide(sprite, group, dokill, collided = None)
```

- 第一个参数指定被检测的精灵。
- 第二个参数指定一个组，由 sprite.Group()生成。
- 第三个参数设置是否从组中删除检测到碰撞的精灵。
- 第四个参数设置一个回调函数，用于定制特殊的检测方法。如果该参数忽略，那么默认是检测精灵之间的 rect 是否产生重叠。

```
# p16_8/main.py
…
    # 用来存放小球对象的列表
    balls = []
    group = pygame.sprite.Group()
    # 创建五个小球
    BALL_NUM = 5
    for i in range(5):
        # 位置随机，速度随机
        position = randint(0, width - 100), randint(0, height - 100)
        speed = [randint(-1, 1), randint(-1, 1)]
        ball = Ball(ball_image, position, speed, bg_size)
```

```
        # 检测新诞生的球是否会卡住其他球
        while pygame.sprite.spritecollide(ball, group, False):
            ball.rect.left, ball.rect.top = randint(0, width - 100),\
            randint(0, height - 100)
        balls.append(ball)
        group.add(ball)
...
        for each in balls:
            each.move()
            screen.blit(each.image, each.rect)
        for each in group:
            # 先从组中移出当前球
            group.remove(each)
            # 判断当前球与其他球是否相撞
            if pygame.sprite.spritecollide(each, group, False):
                each.speed[0] = - each.speed[0]
                each.speed[1] = - each.speed[1]
            # 将当前球添加回组中
            group.add(each)
...
```

结果让人感到沮丧……因为小球有时候竟然在没有碰撞的情况下就弹开了……莫非现成的spritecollide()还不如我们自己写的collide_check()函数精确？

当然不是！之所以会这样，是因为上面的代码没有设置spritecollide()函数的第四个。默认这个参数是None，表示检测的是精灵的rect属性是否重叠，如图16-33所示。

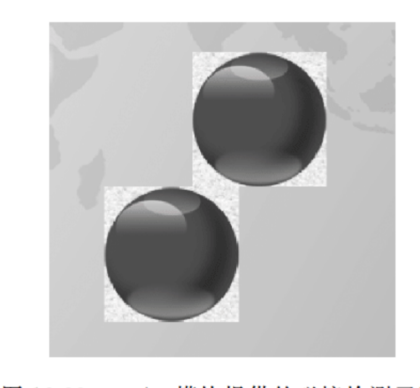


图16-33　sprite模块提供的碰撞检测函数

如果是如图16-33所示的情况，由于小球的背景是透明的，所以看上去就好像没有发生碰撞就弹开了（其实对应的rect已经是重叠了）。因此，需要实现圆形的碰撞检测，还需要指定spritecollide()函数的最后一个参数。

## 16.8.3　实现完美碰撞检测

spritecollide()函数的最后一个参数是指定一个回调函数，用于定制特殊的检测方法。而sprite模块中正好有一个collide_circle()函数用于检测两个圆之间是否发生碰撞。注意：这个函数需要精灵对象中必须有一个radius(半径)属性才行。

```
# p16_8/main.py
class Ball(pygame.sprite.Sprite):
...
        self.radius = self.rect.width / 2
...
    BALL_NUM = 5
    for i in range(BALL_NUM):
        # 位置随机，速度随机
        position = randint(0, width - 100), randint(0, height - 100)
```

```
        speed = [randint(-1, 1), randint(-1, 1)]
        ball = Ball(ball_image, position, speed, bg_size)
        # 检测新诞生的球是否会卡住其他球
        while pygame.sprite.spritecollide(ball, group, False,\
        collide_circle):
            ball.rect.left, ball.rect.top = randint(0, width-100),\
            randint(0, height-100)
        balls.append(ball)
        group.add(ball)
...

        for each in group:
            # 先从组中移出当前球
            group.remove(each)
            # 判断当前球与其他球是否相撞
            if pygame.sprite.spritecollide(each, group, False,\
            collide_circle):
                each.speed[0] = -each.speed[0]
                each.speed[1] = -each.speed[1]
            # 将当前球添加回组中
            group.add(each)
...
```

## 16.9 播放声音和音效



几乎没有任何游戏是一声不吭的，因为多重的感官体验更能刺激玩家的神经。没有声音的游戏就好比是不蘸番茄酱的薯条、忘记带枪的战士……尽管如此，Pygame对于声音的处理并不是特别擅长，我说的是如果你想用Pygame来做一个炫酷的音乐播放器的话可能不行，因为Pygame对声音格式的支持十分有限。不过对于游戏开发来说，是完全足够的。

对于一般游戏来说，声音分为背景音乐和音效两种。背景音乐是时刻伴随着游戏存在的，往往是重复播放的一首歌或曲子；而音效则是在某种条件下被触发产生的，例如两个小球碰撞就会发出啪啪啪的声音。Pygame支持的声音格式十分有限，所以一般情况下用ogg格式作为背景音乐，用无压缩的wav格式作为音效。

播放音效使用mixer模块，需要先生成一个Sound对象，然后调用play()方法来播放。表16-4列举了Sound对象支持的方法。

**表16-4 Sound对象支持的方法**

| 方　　法 | 含　　义 |
|---|---|
| play() | 播放音效 |
| stop() | 停止播放 |
| fadeout() | 淡出 |
| set_volume() | 设置音量 |
| get_volume() | 获取音量 |
| get_num_channels() | 计算该音效播放了多少次 |
| get_length() | 获得该音效的长度 |
| get_raw() | 将该音效以二进制格式的字符串返回 |

播放背景音乐使用 music 模块，music 模块是 mixer 模块中的一个特殊实现，因此使用 pygame. mixer. music 来调用该模块下的方法。表 16-5 列举了 music 模块支持的方法。

**表 16-5　music 模块支持的方法**

| 方　法 | 含　义 | 方　法 | 含　义 |
|---|---|---|---|
| load() | 载入音乐 | get_volume() | 获取音量 |
| play() | 播放音乐 | get_busy() | 检测音乐流是否正在播放 |
| rewind() | 重新播放 | set_pos() | 设置开始播放的位置 |
| stop() | 停止播放 | get_pos() | 获取已经播放的时间 |
| pause() | 暂停播放 | queue() | 将音乐文件放入待播放列表中 |
| unpause() | 恢复播放 | set_endevent() | 在音乐播放完毕时发送事件 |
| fadeout() | 淡出 | get_endevent() | 获取音乐播放完毕时发送的事件类型 |
| set_volume() | 设置音量 | | |

下面编写代码，要求打开程序便开始播放背景音乐(bg_music. ogg)，单击播放 cat. wav 音效，通过右键快捷菜单可播放 dog. wav，利用空格键可暂停/继续播放音乐。

```
# p16_9/music.py
import pygame
import sys
from pygame.locals import *

pygame.init()
pygame.mixer.init() # 初始化混音器模块
# 加载背景音乐
pygame.mixer.music.load("bg_music.ogg")
pygame.mixer.music.set_volume(0.2)
pygame.mixer.music.play()
# 加载音效
cat_sound = pygame.mixer.Sound("cat.wav")
cat_sound.set_volume(0.2)
dog_sound = pygame.mixer.Sound("dog.wav")
dog_sound.set_volume(0.2)
bg_size = width, height = 300, 200
screen = pygame.display.set_mode(bg_size)
pygame.display.set_caption("Music - FishC Demo")
pause = False
pause_image = pygame.image.load("pause.png").convert_alpha()
unpause_image = pygame.image.load("unpause.png").convert_alpha()
pause_rect = pause_image.get_rect()
pause_rect.left, pause_rect.top = (width - pause_rect.width) // 2, \
(height - pause_rect.height) // 2
clock = pygame.time.Clock()

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        if event.type == MOUSEBUTTONDOWN:
            if event.button == 1:
                cat_sound.play()
            if event.button == 3:
```

```
                dog_sound.play()
        if event.type == KEYDOWN:
            if event.key == K_SPACE:
                pause = not pause
    screen.fill((255, 255, 255))
    if pause:
        screen.blit(pause_image, pause_rect)
        pygame.mixer.music.pause()
    else:
        screen.blit(unpause_image, pause_rect)
        pygame.mixer.music.unpause()

    pygame.display.flip()
    clock.tick(30)
```

上面的代码演示了背景声音和音效的使用，现在把声音添加到我们的游戏中：

```
# p16_9/main.py
…
    running = True
    # 添加魔性的背景音乐
    pygame.mixer.music.load('bg_music.ogg')
pygame.mixer.music.play()
    # 添加音效
    loser_sound = pygame.mixer.Sound('loser.wav')
    laugh_sound = pygame.mixer.Sound('laugh.wav')
    winner_sound = pygame.mixer.Sound('winner.wav')
    hole_sound = pygame.mixer.Sound('hole.wav')
…
```

音效只要在需要的时候调用 play()方法即可，而背景音乐我们则希望它能够贯穿游戏的始终。背景音乐完整播放一次视为游戏的时间，因此需要想办法让游戏在背景音乐停止时结束。大家应该有留意到 music 模块有一个 set_endevent()方法，该方法的作用就是在音乐播放完发送一条事件消息。

Pygame 预定义了很多默认的事件，像我们熟悉的键盘事件、鼠标事件等。预定义的事件都有一个标识符，像 MOUSEBUTTONDOWN、KEYDOWN、QUIT 等。其实这些都是一些数字的等值定义，只是为了方便人类理解才做的定义。USEREVENT 以上则是让我们自定义的事件，因此可以像这样自定义事件：

```
MYEVENT1 = USEREVENT
MYEVENT2 = USEREVENT + 1
MYEVENT3 = USEREVENT + 2
…
```

下面的代码让背景音乐播完的时候游戏结束，并播放“失败者”(loser.wav)及“嘲笑”(laugh.wav)的音效：

```
# p16_9/main.py
…
    # 音乐放完时游戏结束!
    GAMEOVER = USEREVENT
pygame.mixer.music.set_endevent(GAMEOVER)
```

```
…
        for event in pygame.event.get():
            if event.type == QUIT:
                sys.exit()
            elif event.type == GAMEOVER:
                loser_sound.play()
                pygame.time.delay(2000)
                laugh_sound.play()
                running = False
…
```

## 16.10 响应鼠标



### 16.10.1 设置鼠标的位置

有了背景音乐、有了小球、有了碰撞检测，接下来需要做的就是设计“摩擦摩擦”的代码了。这有一块玻璃面板的图片，如图 16-34 所示。

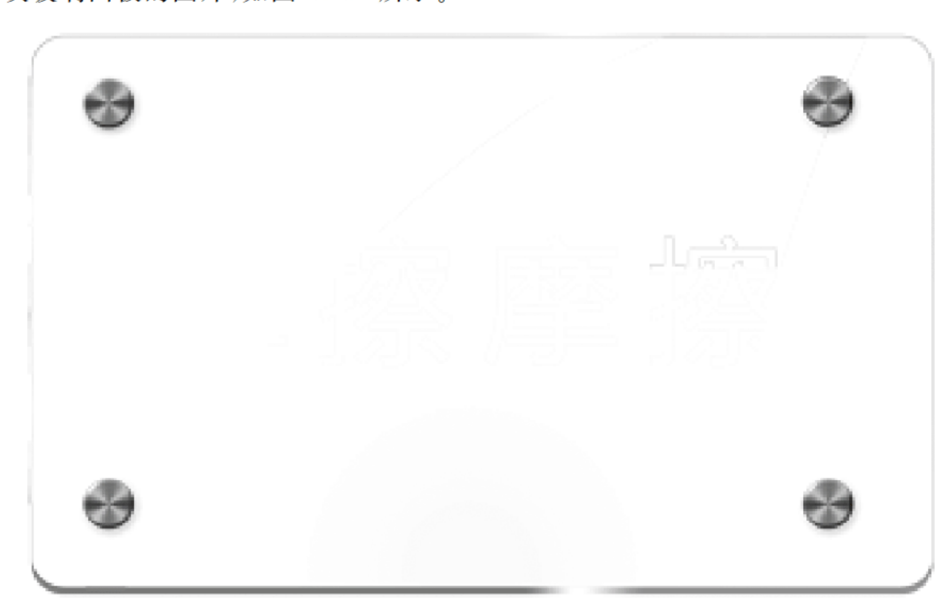


图 16-34 游戏素材

把它放在游戏界面的下方位置，并限制鼠标只能在里边移动。一步一步来，先创建一个 Glass 类，用于表示这块玻璃：

```
…
class Glass(pygame.sprite.Sprite):
    def __init__(self, glass_image, bg_size):
        pygame.sprite.Sprite.__init__(self)
        self.glass_image = pygame.image.load(glass_image).convert_alpha()
        self.glass_rect = self.glass_image.get_rect()
        self.glass_rect.left, self.glass_rect.top = \
                              (bg_size[0] - self.glass_rect.width) // 2, \
```

```
                    bg_size[1] - self.glass_rect.height
...
    # 生成用于摩擦摩擦的玻璃面板
area = Glass(glass_image, bg_size)
...
        screen.blit(background, (0, 0))
        # 绘制用于摩擦摩擦的玻璃面板
        screen.blit(area.glass_image, area.glass_rect)
...
```

代码实现如图 16-35 所示。




图 16-35　游戏界面

下一步是限制鼠标只能在玻璃面板中移动，并使用一个小手的图案代替原来的鼠标光标等，你说限制鼠标移动？说的容易，鼠标怎么移动是玩家的事，我还能干预它？

当然可以，程序是你写的，在你的地盘上当然是你做主！可以先通过 mouse 模块的 get_pos()方法获取鼠标的当前位置，检测如果超出了玻璃面板的范围，则使用 set_pos()修改它。

## 16.10.2　自定义鼠标光标

作为一个游戏，当然希望鼠标的光标可以更漂亮一些，所以需要替换掉原来"黑土小"的箭头光标。这里直接用一个小手的图片来替换掉原来的光标。做法就是使用 mouse 模块的 set_visible()方法将原来的光标设置为"不可见"，然后在鼠标的当前位置上绘制小手的图片。

代码实现如下：

```
...
class Glass(pygame.sprite.Sprite):
    def __init__(self, glass_image, mouse_image, bg_size):
```

```
        …
        self.mouse_image = pygame.image.load(mouse_image).convert_alpha()
        self.mouse_rect = self.mouse_image.get_rect()
        self.mouse_rect.left, self.mouse_rect.top = \
        self.glass_rect.left, self.glass_rect.top
        # 初始化鼠标的位置于左上角
        pygame.mouse.set_pos([self.glass_rect.left, self.glass_rect.top])
        # 鼠标不可见
        pygame.mouse.set_visible(False)
…
        screen.blit(background, (0, 0))
        screen.blit(area.glass_image, area.glass_rect)
        # 获取鼠标的当前位置,并设置代替光标的图片
        area.mouse_rect.left, area.mouse_rect.top = pygame.mouse.get_pos()
        # 限制鼠标只能在玻璃内摩擦摩擦
        if area.mouse_rect.left < area.glass_rect.left:
            area.mouse_rect.left = area.glass_rect.left
        if area.mouse_rect.left > area.glass_rect.right - \
        area.mouse_rect.width:
            area.mouse_rect.left = area.glass_rect.right - \
            area.mouse_rect.width
        if area.mouse_rect.top < area.glass_rect.top:
            area.mouse_rect.top = area.glass_rect.top
        if area.mouse_rect.top > area.glass_rect.bottom - \
        area.mouse_rect.height:
            area.mouse_rect.top = area.glass_rect.bottom - \
            area.mouse_rect.height
        screen.blit(area.mouse_image, area.mouse_rect)
```

代码实现如图 16-36 所示。




图 16-36　替换掉原来的鼠标

### 16.10.3 让小球响应光标的移动频率

接下来要让小球可以响应鼠标的“摩擦”，当鼠标的移送速度符合某个频率段时，小球将停下来并变成绿色。

大家知道鼠标的移动会不断产生事件，所以可以利用这一点，让每一个小球响应1秒钟时间内不同数量的事件。

做法如下：

(1) 为每个小球设定一个不同的目标；

(2) 创建一个motion变量来记录鼠标每1秒钟产生事件数量；

(3) 为小球添加一个check()方法，用于判断鼠标在1秒钟时间内产生的事件数量是否匹配此目标；

(4) 添加一个自定义事件，每1秒钟触发1次。调用每个小球的check()检测是motion的值是否匹配某一个小球的目标，并将motion重新初始化，以便记录下1秒的鼠标事件数量；

(5) 小球应该添加一个control属性，用于记录当前的状态(绿色->玩家控制 or 灰色->随机移动)；

(6) 通过检查control属性决定绘制什么颜色的小球。

代码实现如下：

```
…
# p16_10/main.py
class Ball(pygame.sprite.Sprite):
    def __init__(self, grayball_image, greenball_image, position, \
    speed, bg_size, target):
        # 初始化动画精灵
        pygame.sprite.Sprite.__init__(self)
        self.grayball_image = \
        pygame.image.load(grayball_image).convert_alpha()
        self.greenball_image = \
        pygame.image.load(greenball_image).convert_alpha()
        self.rect = self.grayball_image.get_rect()
        # 将小球放在指定位置
        self.rect.left, self.rect.top = position
        self.radius = self.rect.width / 2
        self.width, self.height = bg_size[0], bg_size[1]
        self.speed = speed
        self.target = target
        self.control = False


    def check(self, motion):
        # 要求100%匹配是很难的，所以还是降低点难度吧
```

```
            if self.target < motion < self.target + 5:
                return True
            else:
                return False
...
    # 创建五个小球
    BALL_NUM = 5
    for i in range(BALL_NUM):
        # 位置随机,速度随机
        position = randint(0, width-100), randint(0, height-100)
        speed = [randint(-10, 10), randint(-10, 10)]
        ball = Ball(grayball_image, greenball_image, position, \
        speed, bg_size, 5 * (i+1))
...
    # 生成用于摩擦摩擦的玻璃面板
    area = Glass(glass_image, mouse_image, bg_size)
    # motion 记录鼠标在玻璃面板产生的事件数量
    motion = 0
    # 1 秒检查一次摩擦摩擦
    MYTIMER = USEREVENT + 1
    pygame.time.set_timer(MYTIMER, 1000)
clock = pygame.time.Clock()
...
        for event in pygame.event.get():
            ...
            elif event.type == MOUSEMOTION:
                motion += 1

            elif event.type == MYTIMER:
                if motion:
                    for each in group:
                        if each.check(motion):
                            each.speed = [0, 0]
                            each.control = True
                    motion = 0
...
        for each in balls:
            each.move()
            if each.control:
                screen.blit(each.greenball_image, each.rect)
            else:
                screen.blit(each.grayball_image, each.rect)
...
```

程序实现如图 16-37 所示。

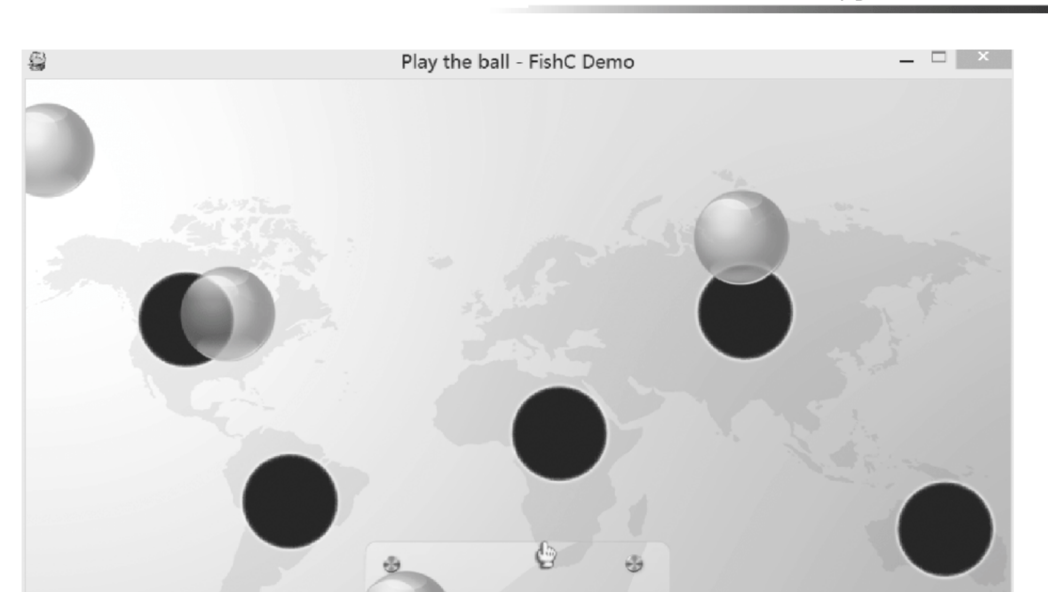



图 16-37 让小球响应光标的移动频率



## 16.11 响应键盘

通过"摩擦摩擦"可以使小球变绿色，玩家此时可以通过键盘上的 W、S、A、D 按键上下左右地移动小球。下边代码响应相应的键盘事件：

```
…
elif event.type == KEYDOWN:
    if event.key == K_w:
        for each in group:
            if each.control:
                each.speed[1] -= 1
    if event.key == K_s:
        for each in group:
            if each.control:
                each.speed[1] += 1
    if event.key == K_a:
        for each in group:
            if each.control:
                each.speed[0] -= 1
    if event.key == K_d:
        for each in group:
            if each.control:
                each.speed[0] += 1
…
```

程序执行后，无论玩家是短暂地按下按键还是持续紧按，结果都只是让小球以龟速移动，

并没有实现所谓"带加速度的快感"。这是由于默认情况下,无论你是简单的按一下按键还是紧按着不松开,Pygame 都只为你发送一个键盘按下的事件。不过事实上可以通过 key 模块的 set_repeat()方法,来设置是否重复响应持续按下某个按键。

```
set_repeat(delay, interval)
```

- delay 参数指定第一次发送事件的延迟时间
- interval 参数指定重复发送事件的时间间隔
- 如果不带任何参数,表示取消重复发送事件

为了使得小球获得加速度的快感,设置按键的重复响应间隔为 100 毫秒:

```
…
# 设置持续按下键盘的重复响应
pygame.key.set_repeat(100, 100)
…
```

小球在碰撞后失去控制,只需要在检测到碰撞时将 control 属性改为 False,小球即脱离控制:

```
…
if pygame.sprite.spritecollide(each, group, False, \
pygame.sprite.collide_circle):
    each.speed[0] = - each.speed[0]
    each.speed[1] = - each.speed[1]
    each.control = False
…
```

## 16.12 结束游戏

### 16.12.1 发生碰撞后获得随机速度

添加了上面的代码,绿色的小球已经能听凭你的使唤了。接下来要做的就是让小球在碰撞的时候获得一个新的随机速度,这将加大游戏的难度。

```
…
for each in group:
    # 先从组中移出当前球
    group.remove(each)
    # 判断当前球与其他球是否相撞
    if pygame.sprite.spritecollide(each, group, False, \
    pygame.sprite.collide_circle):
        each.speed = [randint( - 10, 10), randint( - 10, 10)]
        each.control = False
    # 将当前球添加回组中
    group.add(each)
…
```

但是程序实现后……意想不到的事情发生了,如图 16-38 所示。

两个小球碰撞的时候经常会发生"抖动"现象,浪漫的读者朋友看到的可能是俩小球如胶似漆,彼此不愿分开的缠绵……而事实上更多玩家看到的则是:这游戏怎么这么卡?






图 16-38 出现 bug

## 16.12.2 减少"抖动"现象的发生

分析一下出现"抖动"的原因，无非就是每次碰撞都获得一个随机的速度导致。由于随机的速度带有方向（负数往左，正数往右），所以如果两个小球刚好得到的速度方向是相向的，那么就会再次发生碰撞，直到速度方向为反向，并且一次移动的距离可以彼此分开为止。

解决这个问题的方法就是将方向和速度两个概念独立开来，因此为小球添加一个 side 属性用于表示方向，－1 表示向左，1 表示向右。然后在每次检测到碰撞的时候先将方向取反，以相同的速度反向移动一次后，再重新获取随机速度。

先给小球添加一个 side 属性和一个 collide 属性，collide 属性用于标志是否发生碰撞，如果发生碰撞，再下一次的移动后获得随机速度：

```
class Ball(pygame.sprite.Sprite):
def __init__(self, image1, image2, position, speed, bg_size, level):
    …
        self.side = [choice([-1, 1]), choice([-1, 1])]
        self.collide = False
    …
```

既然将原来带方向的速度拆分为方向和速度两个属性，那么小球的自由移动就应该由两个属性相乘得到：

```
class Ball(pygame.sprite.Sprite):
    …
    def move(self):
        self.rect = self.rect.move((self.side[0] * self.speed[0], \
```

```
        self.side[1] * self.speed[1]))
    …
```

由于速度不再表示方向，所以随机速度不应该存在负数：

```
…
BALL_NUM = 5
for i in range(BALL_NUM):
    # 位置随机，速度随机
    position = randint(0, width - 100), randint(0, height - 100)
    speed = [randint(1, 10), randint(1, 10)]
…
```

碰撞发生时，首先修改的是方向：

```
…
if pygame.sprite.spritecollide(each, group, False, \
pygame.sprite.collide_circle):
    each.side[0] = - each.side[0]
    each.side[1] = - each.side[1]
    each.collide = True
    each.control = False
…
```

进行一次移动后再获取随机速度：

```
…
for each in balls:
    each.move()
    if each.collide:
        each.speed = [randint(1, 10), randint(1, 10)]
        each.collide = False
…
```

程序运行后新的 BUG 又出现了，控制权交到玩家手上时，小球并不能正确地按照玩家的操作去移动……现实中的开发常常会碰到这样的情景，补完一个 BUG 或新添加一个功能，直接影响了原来正确的代码逻辑，导致另一个 BUG 的出现。

为什么会导致玩家的操作无法正确地控制小球呢？仔细检查代码之后发现原来将带方向的速度拆分为方向和速度，而响应玩家按键操作的代码仍旧认为速度是带方向的（如果速度为负数，方向为负数，那么得到的却是反方向的移动）。

为了保留玩家操控小球是带加速度的这一特性，不妨将小球的移动给区分开：

```
…
def move(self):
    if self.control:
        self.rect = self.rect.move(self.speed)
    else:
        self.rect = self.rect.move( \
        (self.side[0] * self.speed[0], self.side[1] * self.speed[1]))
…
```

小球发生碰撞失去控制，将带方向的速度拆分：

```
…
if pygame.sprite.spritecollide(each, group, False, \
pygame.sprite.collide_circle):
    each.side[0] = - each.side[0]
    each.side[1] = - each.side[1]
    each.collide = True
    if each.control:
        each.side[0] = -1
        each.side[1] = -1
        each.control = False
…
```

这么改完之后发现抖动的现象有了显著减少，只是偶尔两个小球会卡在边框之外。所以这里再修改一下 move()限制边界的范围：

```
…
def move(self):
    if self.control:
        self.rect = self.rect.move(self.speed)
    else:
        self.rect = self.rect.move( \
        (self.side[0] * self.speed[0], self.side[1] * self.speed[1]))
    # 如果小球的左侧出了边界,那么将小球左侧的位置改为右侧的边界
    # 这样便实现了从左边进入,右边出来的效果
    if self.rect.right < 0:
        self.rect.left = self.width
    elif self.rect.left > self.width:
        self.rect.right = 0
    elif self.rect.bottom < 0:
        self.rect.top = self.height
    elif self.rect.top > self.height:
        self.rect.bottom = 0
…
```

### 16.12.3 游戏胜利

当绿色的小球移动到黑洞的正上方时，只要玩家立刻敲下键盘的空格键，那么小球将被"填"入到黑洞中。此后其他小球将直接从其上方飘过，无视它的存在。音乐结束前，如果所有的小球都被填入到各个黑洞中，游戏胜利。

这里有两点需要注意：第一是每个黑洞只能填入一个绿色的小球；第二是当小球填入黑洞时，其他小球会从其上方飘过，而不是下方。

首先，将五个黑洞的位置定义好：

```
# 五个黑洞的范围,因为 100% 命中太难,所以只要在范围内即可
# 每个元素:(x1, x2, y1, y2)
hole = [(117, 119, 199, 201), (225, 227, 390, 392), (503, 505, 320, 322), (698, 700, 192, 194),
(906, 908, 419, 421)]
```

当玩家按下空格键时，检测每个小球的当前位置是否匹配任何一个黑洞的范围，如果是，那么固定它。如果所有的黑洞都被补上，游戏胜利：

```
if event.key == K_SPACE:
    # 判断小球是否在坑内
    for each in group:
        if each.moving:
            for i in hole:
                if i[0] <= each.rect.left <= i[1] and i[2] \
                <= each.rect.top <= i[3]:
                    # 播放音效
                    hole_sound.play()
                    each.speed = [0, 0]
                    # 从 group 中移出,这样其他球就会忽视它
                    group.remove(each)
                    # 放到 balls 列表中的最前,也就是第一个绘制的球
                    # 这样当球在坑里时,其他球会从它上边过去,而不是下边
                    temp = balls.pop(balls.index(each))
                    balls.insert(0, temp)
                    # 一个坑一个球
                    hole.remove(i)
            # 坑都补完了,游戏结束
            if not hole:
                pygame.mixer.music.stop()
                # 播放胜利配乐
                winner_sound.play()
                pygame.time.delay(3000)
                # 打印然并卵
                msg = pygame.image.load("win.png").convert_alpha()
                msg_pos = (width - msg.get_width()) // 2, \
                (height - msg.get_height()) // 2
                msgs.append((msg, msg_pos))
                # 播放嘲笑
                laugh_sound.play()
```

### 16.12.4 更好地结束游戏

为了在 IDLE 下单击关闭按钮可以正常结束游戏,可以在响应 QUIT 事件的时候先调用 pygame.quit():

```
if event.type == QUIT:
    pygame.quit()
    sys.exit()
```

如果用户双击打开游戏的文件,那么如果有逻辑错误或者代码错误,程序可能就会直接关闭。这样对于调试也是不利的,因此可以这么改:

```
if __name__ == "__main__":
    # 这样做的好处是双击打开时如果出现异常可以报告异常,而不是一闪而过!
    try:
        main()
    except SystemExit:
        pass
    except:
```

```
        traceback.print_exc()
        # 释放已经初始化的资源
        pygame.quit()
        input()
```

完整实现代码见附件 P16_11。

## 16.13 经典飞机大战



不知道大家有没有打过飞机，喜不喜欢打飞机。当我第一次接触这个东西的时候，我的内心是被震撼到的。第一次接触打飞机的时候作者本人是身心愉悦的，因为周边的朋友都在打飞机，每次都会下意识彼此较量一下，看谁打得更好。打飞机也是需要有一定的技巧的，熟练的朋友一把能打上半个小时，生疏的则三五分钟就败下阵来。

### 16.13.1 游戏设定

游戏界面如图 16-39～图 16-41 所示。

游戏的基本设定：

- 敌方共有大中小 3 款飞机，分为高中低三种速度；
- 子弹的射程并非全屏，而大概是屏幕长度的 80%；
- 消灭小飞机需要 1 发子弹，中飞机需要 8 发，大飞机需要 20 发子弹；
- 每消灭一架小飞机得 1000 分，中飞机 6000 分，大飞机 10000 分；
- 每隔 30 秒有一个随机的道具补给，分为两种道具，全屏炸弹和双倍子弹；
- 全屏炸弹最多只能存放 3 枚，双倍子弹可以维持 18 秒钟的效果；
- 游戏将根据分数来逐步提高难度，难度的提高表现为飞机数量的增多以及速度的加快。

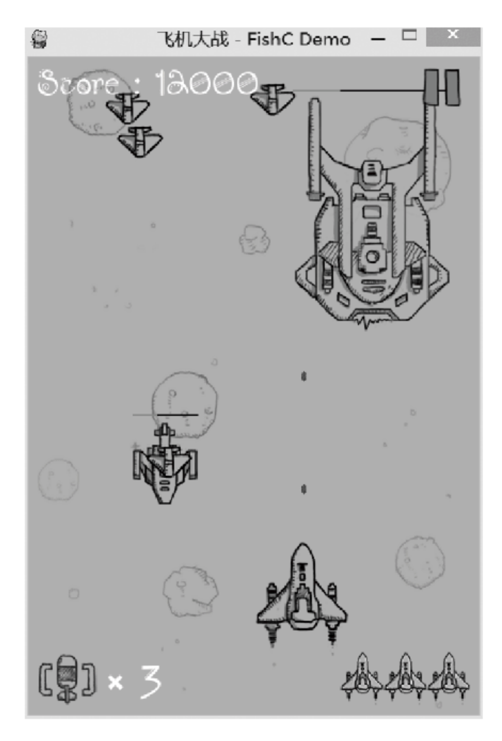



图 16-39　打飞机游戏(一)

另外还对游戏做了一些改进，比如为中飞机和大飞机增加了血槽的显示，这样玩家可以直观地知道敌机快被消灭了没有；我方有三次机会，每次被敌人消灭，新诞生的飞机会有 3 秒钟的安全期；游戏结束后会显示历史最高分数。

这个游戏加上基本的注释代码量在 800 行左右，代码看上去比较多，主要是作者本人奉行着“多大代码少动脑”的开发原则。所以大家不要怕，越是多的代码，逻辑就越容易看得清楚，就越好学习。好，那让我们从无到有，从简单到复杂来一起打造这个游戏吧！完整代码及资源可参考附件：p16_13。

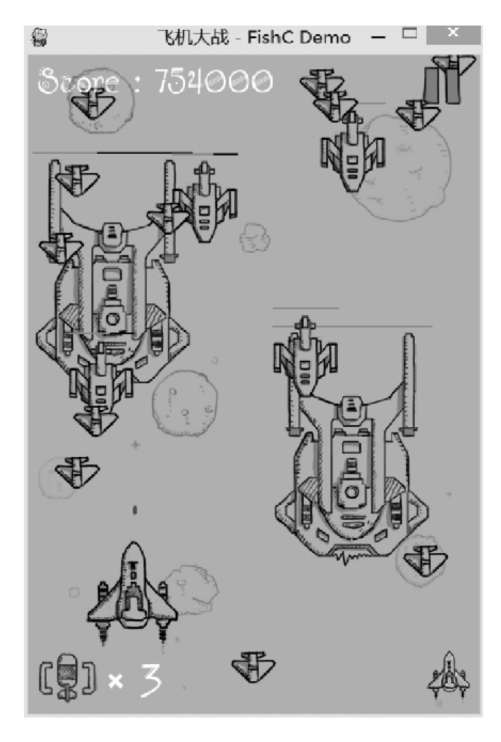



图 16-40　打飞机游戏(二)




图 16-41　打飞机游戏(三)

首先,把能够独立开的代码独立成模块:

- main.py——主模块。
- myplane.py——定义我方飞机。
- enemy.py——定义敌方飞机。
- bullet.py——定义子弹。
- supply.py——定义补给。

资源文件分类存放:

- sound ——声音、音效资源。
- images ——图片资源。
- font ——字体资源。

## 16.13.2　主模块

先写主模块的代码:

```
# main.py
import pygame
import sys
import traceback
import myplane
import bullet
import enemy
import supply
from pygame.locals import *
```

```
from random import *

pygame.init()
pygame.mixer.init()
bg_size = width, height = 480, 700
screen = pygame.display.set_mode(bg_size)
pygame.display.set_caption("飞机大战 -- FishC Demo")
background = pygame.image.load("images/background.png").convert()
# 载入游戏音乐
pygame.mixer.music.load("sound/game_music.ogg")
pygame.mixer.music.set_volume(0.2)
bullet_sound = pygame.mixer.Sound("sound/bullet.wav")
bullet_sound.set_volume(0.2)
bomb_sound = pygame.mixer.Sound("sound/use_bomb.wav")
bomb_sound.set_volume(0.2)
supply_sound = pygame.mixer.Sound("sound/supply.wav")
supply_sound.set_volume(0.2)
get_bomb_sound = pygame.mixer.Sound("sound/get_bomb.wav")
get_bomb_sound.set_volume(0.2)
get_bullet_sound = pygame.mixer.Sound("sound/get_bullet.wav")
get_bullet_sound.set_volume(0.2)
upgrade_sound = pygame.mixer.Sound("sound/upgrade.wav")
upgrade_sound.set_volume(0.2)
enemy3_fly_sound = pygame.mixer.Sound("sound/enemy3_flying.wav")
enemy3_fly_sound.set_volume(0.2)
enemy1_down_sound = pygame.mixer.Sound("sound/enemy1_down.wav")
enemy1_down_sound.set_volume(0.1)
enemy2_down_sound = pygame.mixer.Sound("sound/enemy2_down.wav")
enemy2_down_sound.set_volume(0.2)
enemy3_down_sound = pygame.mixer.Sound("sound/enemy3_down.wav")
enemy3_down_sound.set_volume(0.5)
me_down_sound = pygame.mixer.Sound("sound/me_down.wav")
me_down_sound.set_volume(0.2)

def main():
    pygame.mixer.music.play(-1)
    clock = pygame.time.Clock()
    running = True

    while running:
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()
        screen.blit(background, (0, 0))
        pygame.display.flip()
        clock.tick(60)

if __name__ == "__main__":
    try:
        main()
    except SystemExit:
```

```
        pass
    except:
        traceback.print_exc()
        pygame.quit()
        input()
```

## 16.13.3 我方飞机



接下来应该让主角登场，创建一个 myplane.py 模块来定义我方飞机：

```
# myplane.py
import pygame

class MyPlane(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        pygame.sprite.Sprite.__init__(self)
        self.image = \ pygame.image.load("images/me1.png").convert_alpha()
        self.rect = self.image.get_rect()
        self.width, self.height = bg_size[0], bg_size[1]
        # 初始化位于下方的中间位置
        # 下方预留 60 像素左右的位置作为"状态栏"
        self.rect.left, self.rect.top = (self.width - \ self.rect.width)\
        // 2, self.height - self.rect.height - 60
        self.speed = 10
```

分别定义 moveUp()、moveDown()、moveLeft()和 moveRight()控制我方飞机上、下、左、右移动：

```
def moveUp(self):
    if self.rect.top > 0:
        self.rect.top -= self.speed
    else:
        self.rect.top = 0
def moveDown(self):
    if self.rect.bottom < self.height - 60:
        self.rect.top += self.speed
    else:
        self.rect.bottom = self.height - 60
def moveLeft(self):
    if self.rect.left > 0:
        self.rect.left -= self.speed
    else:
        self.rect.left = 0
def moveRight(self):
    if self.rect.right < self.width:
        self.rect.left += self.speed
    else:
        self.rect.right = self.width
```

## 16.13.4 响应键盘

接着需要在 main 模块中响应用户的键盘操作。响应用户的键盘操作有两种方法：第一种是通过 KEYDOWN 或 KEYUP 事件得知用户是否按下键盘按键；第二种是调用 key 模块

的 get_pressed()方法，它会返回一个序列，包含当前键盘上所有按键的状态。

对于检测偶尔触发的键盘事件，推荐使用第一种方法。但对于频繁触发的键盘事件，建议使用第二种方法。由于整个游戏就是通过键盘来控制我方飞机，所以毅然决然选用第二种方法：

```
…
# 检测用户的键盘操作
key_pressed = pygame.key.get_pressed()
# 移动我方飞机
if key_pressed[K_w] or key_pressed[K_UP]:
    me.moveUp()
if key_pressed[K_s] or key_pressed[K_DOWN]:
    me.moveDown()
if key_pressed[K_a] or key_pressed[K_LEFT]:
    me.moveLeft()
if key_pressed[K_d] or key_pressed[K_RIGHT]:
    me.moveRight()
screen.blit(background, (0, 0))
# 绘制我方飞机
screen.blit(me.image, me.rect)
…
```

### 16.13.5 飞行效果

为了增加我方飞机的动态效果，可以通过下边两张图片的不断切换来实现飞机“突突突”的飞行效果：

```
# myplane.py
class MyPlane(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        pygame.sprite.Sprite.__init__(self)
        self.image1 = pygame.image.load("images/me1.png").convert_alpha()
        self.image2 = pygame.image.load("images/me2.png").convert_alpha()
        self.rect = self.image1.get_rect()

# main.py
…
switch_image = True
…
while running:
…
    switch_image = not switch_image
    # 绘制我方飞机
    if switch_image:
        screen.blit(me.image1, me.rect)
    else:
        screen.blit(me.image2, me.rect)
…
```

但实现起来效果并不理想，因为切换的速度太快了……所以必须想办法在不影响游戏正

常运行的条件下增加点“延迟”才行。这里可以使用单片机开发中很常用的一招——设置延时变量：

```
# main.py
…
delay = 100
…
while running:
    …
    # 切换图片
    if not(delay % 5):
        switch_image = not switch_image
    delay -= 1
    if not delay:
        delay = 100
…
```

现在我方飞机的画面就是5帧切换一次，如果限定帧率为60，则一秒钟最多切换12次。

### 16.13.6 敌方飞机

既然英雄已经有了，那现在就是需要创造敌人的时候。敌机分为小、中、大三个尺寸，它们的速度依次是快、中、慢，在游戏界面的上方位置创造位置随机的敌机，可以让它们不在同一排出现。将敌机的定义写在enemy.py模块中：

```
# enemy.py
import pygame
from random import *

class SmallEnemy(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        pygame.sprite.Sprite.__init__(self)

        self.image = \
        pygame.image.load("images/enemy1.png").convert_alpha()
        self.rect = self.image.get_rect()
        self.width, self.height = bg_size[0], bg_size[1]
        self.speed = 2
        self.rect.left, self.rect.bottom = \
                randint(0, self.width - self.rect.width),\
                randint(-5 * self.height, 0)
```

由于敌机只会一个劲儿地往前冲，所以敌机的移动只是简单地增加rect.top的值，当敌机的坐标超出屏幕底端，则修改rect.top的位置，让它重新出现在屏幕上方的随机位置：

```
…
    def move(self):
        if self.rect.top < self.height:
            self.rect.top += self.speed
        else:
            self.reset()
```

```
    def reset(self):
        self.rect.left, self.rect.bottom = \
                randint(0, self.width - self.rect.width), \
                randint(-5 * self.height, 0)
...
```

这是小型敌机，同样的方法可以定义出中、大型敌机。其中大型敌机作为BOSS级别的存在，它的飞行也是有特写和音效的。另外对比起小型敌机的普遍存在，中、大型敌机显得会更少一些，因此将生成的随机位置扩大范围：

```
class MidEnemy(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        ...
        self.speed = 1
        self.rect.left, self.rect.bottom = \
                randint(0, self.width - self.rect.width), \
                randint(-10 * self.height, -self.height)

class BigEnemy(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        ...
        self.image1 = \
        pygame.image.load("images/enemy3_n1.png").convert_alpha()
        self.image2 = \
        pygame.image.load("images/enemy3_n2.png").convert_alpha()
        ...
        self.speed = 1
        self.rect.left, self.rect.bottom = \
                randint(0, self.width - self.rect.width), \
                randint(-15 * self.height, -5 * self.height)
```

敌机的定义有了，接下来就是要在main模块中实例化出来：

```
# main.py
...
def main():
    ...
    enemies = pygame.sprite.Group()
    # 生成敌方小型飞机
    small_enemies = pygame.sprite.Group()
    add_small_enemies(small_enemies, enemies, 15)
    # 生成敌方中型飞机
    mid_enemies = pygame.sprite.Group()
    add_mid_enemies(mid_enemies, enemies, 4)
    # 生成敌方大型飞机
    big_enemies = pygame.sprite.Group()
    add_big_enemies(big_enemies, enemies, 2)
...
```

### 16.13.7 提升敌机速度

随着分数越来越高，游戏难度会逐渐提升。难度的提升主要表现在敌机数量的增加和速

度的加快。所以将添加敌机写成一个函数，方便以后调用：

```
# main.py
...
def add_small_enemies(group1, group2, num):
    for i in range(num):
        e1 = enemy.SmallEnemy(bg_size)
        group1.add(e1)
        group2.add(e1)

def add_mid_enemies(group1, group2, num):
    for i in range(num):
        e2 = enemy.MidEnemy(bg_size)
        group1.add(e2)
        group2.add(e2)

def add_big_enemies(group1, group2, num):
    for i in range(num):
        e3 = enemy.BigEnemy(bg_size)
        group1.add(e3)
        group2.add(e3)
...
```

让敌机在界面上飞一会儿：

```
# main.py
...
def main():
    ...
    # 绘制大型敌机
    for each in big_enemies:
        each.move()
        if switch_image:
            screen.blit(each.image1, each.rect)
        else:
            screen.blit(each.image2, each.rect)
        # 即将出现在画面中,播放音效
        if each.rect.bottom > -50:
            enemy3_fly_sound.play()
    # 绘制中型敌机
    for each in mid_enemies:
        each.move()
        screen.blit(each.image, each.rect)
    # 绘制小型敌机
    for each in small_enemies:
        each.move()
        screen.blit(each.image, each.rect)
...
```

## 16.13.8 碰撞检测



当敌我两机发生碰撞的时候，双方应该是玉石俱焚的。现在为每个类添加撞

机发生时的惨烈画面：

```
# myplane.py
class MyPlane(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        …
        self.destroy_images = []
        self.destroy_images.extend([\
pygame.image.load("images/me_destroy_1.png").convert_alpha(),\
pygame.image.load("images/me_destroy_2.png").convert_alpha(),\
pygame.image.load("images/me_destroy_3.png").convert_alpha(),\
pygame.image.load("images/me_destroy_4.png").convert_alpha() \
            ])
…

# enemy.py
class SmallEnemy(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        …
        self.destroy_images = []
        self.destroy_images.extend([\
pygame.image.load("images/enemy1_down1.png").convert_alpha(),\
pygame.image.load("images/enemy1_down2.png").convert_alpha(),\
pygame.image.load("images/enemy1_down3.png").convert_alpha(),\
pygame.image.load("images/enemy1_down4.png").convert_alpha() \
            ])
…
class MidEnemy(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        …
        self.destroy_images = []
        self.destroy_images.extend([\
pygame.image.load("images/enemy2_down1.png").convert_alpha(),\
pygame.image.load("images/enemy2_down2.png").convert_alpha(),\
pygame.image.load("images/enemy2_down3.png").convert_alpha(),\
pygame.image.load("images/enemy2_down4.png").convert_alpha() \
            ])
…
class BigEnemy(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        …
        self.destroy_images = []
        self.destroy_images.extend([\
        pygame.image.load("images/enemy3_down1.png").convert_alpha(),\
        pygame.image.load("images/enemy3_down2.png").convert_alpha(),\
        pygame.image.load("images/enemy3_down3.png").convert_alpha(),\
        pygame.image.load("images/enemy3_down4.png").convert_alpha(),\
        pygame.image.load("images/enemy3_down5.png").convert_alpha(),\
        pygame.image.load("images/enemy3_down6.png").convert_alpha() \
            ])
…
```

然后为每个类添加一个 active 属性，当该属性为 True 表示飞机正常飞行，否则表示已经遇难，显示毁灭图片：

```
# main.py
…
def main():
    …
    # 中弹图片索引
    e1_destroy_index = 0
    e2_destroy_index = 0
    e3_destroy_index = 0
    me_destroy_index = 0
while running:
        …
        # 绘制大型敌机
        for each in big_enemies:
            if each.active:
                each.move()
                if switch_image:
                    screen.blit(each.image1, each.rect)
                else:
                    screen.blit(each.image2, each.rect)
                # 即将出现在画面中，播放音效
                if each.rect.bottom > -50:
                    enemy3_fly_sound.play()
            else:
                # 毁灭
                enemy3_down_sound.play()
                if not(delay % 3):
                    screen.blit(each.destroy_images[e3_destroy_index],\
                    each.rect)
                    e3_destroy_index = (e3_destroy_index + 1) % 6
                    if e3_destroy_index == 0:
                        each.reset()
        # 绘制中型敌机:
        for each in mid_enemies:
            if each.active:
                each.move()
                screen.blit(each.image, each.rect)
            else:
                # 毁灭
                enemy2_down_sound.play()
                if not(delay % 3):
                    screen.blit(each.destroy_images[e2_destroy_index],\
                    each.rect)
                    e2_destroy_index = (e2_destroy_index + 1) % 4
                    if e2_destroy_index == 0:
                        each.reset()
        # 绘制小型敌机:
        for each in small_enemies:
            if each.active:
                each.move()
                screen.blit(each.image, each.rect)
            else:
                # 毁灭
```

```
                enemy1_down_sound.play()
                if not(delay % 3):
                    screen.blit(each.destroy_images[e1_destroy_index],\
                    each.rect)
                    e1_destroy_index = (e1_destroy_index + 1) % 4
                    if e1_destroy_index == 0:
                        each.reset()
        # 绘制我方飞机
        if me.active:
            if switch_image:
                screen.blit(me.image1, me.rect)
            else:
                screen.blit(me.image2, me.rect)
        else:
            # 毁灭
            me_down_sound.play()
            if not(delay % 3):
                screen.blit(me.destroy_images[me_destroy_index], me.rect)
                me_destroy_index = (me_destroy_index + 1) % 4
                if me_destroy_index == 0:
                    print("Game Over!")
                    running = False
...
```

下面写碰撞检测代码，一旦我方飞机碰撞到敌机，导致的结果就是敌我双方同归于尽：

```
...
while running:
    ...
    # 检测我方飞机是否被撞
    enemies_down = pygame.sprite.spritecollide(me, enemies, False)
    if enemies_down:
        me.active = False
        for e in enemies_down:
            e.active = False
...
```

## 16.13.9 完美碰撞检测

由于前边只是使用普通的 spritecollide()函数进行碰撞检测，所以默认是以图片的矩形区域作为检测范围，因此看到的是两飞机并没有真正相撞就都毁了，如图 16-42 所示。

其实 Pygame 是可以做到完美碰撞检测的。sprite 模块中有个 collide_mask()函数可以利用，该函数要求检测的对象拥有一个叫作 mask 的属性，用于指定检测的范围。关于 mask，Pygame 还专门整了个 mask 模块，其中的 from_surface()函数可以将一个 Surface 对象中的非透明部分标志位 mask 并返回。

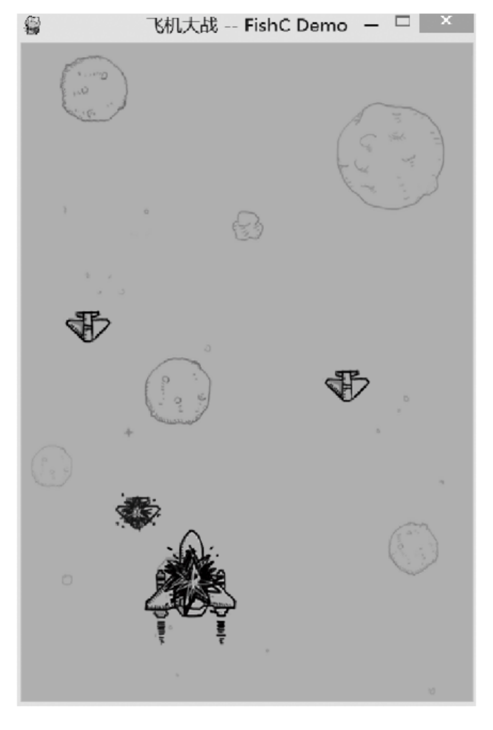
依葫芦画瓢，在敌机和我方飞机的类定义中加入：

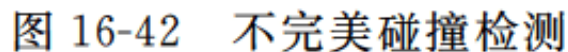
```
self.mask = pygame.mask.from_surface(self.image)
```

然后将检测碰撞的函数改为：

```
enemies_down = pygame.sprite.spritecollide(me, enemies, False, pygame.sprite.collide_mask)
```

这就实现了完美碰撞检测，如图 16-43 所示。





图 16-42　不完美碰撞检测

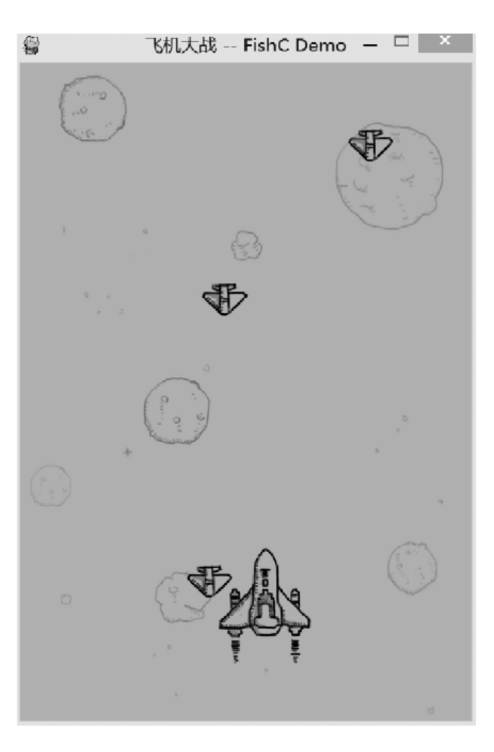



图 16-43　完美碰撞检测

## 16.13.10　一个 BUG

细心的读者朋友应该不难发现，刚才的代码其实有一个明显的 BUG，导致部分音效无法正常播放。不继续往下看，你能自己找出来吗？

无论是敌机还是我方飞机，当它们毁灭的时候，播放音效的代码是这么被执行的：

```
…
if each.active:
    …
else:
# 毁灭
# 播放飞机毁灭音效
    if not(delay % 3):
        …
…
```

这样写有什么问题吗？当然有！你看，一个飞机毁灭只需要播放一次音效，但飞机毁灭的画面并不止一帧，导致重复地播放多次同一个毁灭的音效，同时占用了很多播放音效的通道，而 Pygame 默认却只有八条通道。可想而知，当很多音效同时需要播放时，后边的音效就没有空闲的通道可以播放了。

所以解决方案就是让每个音效只播放一次：

```
…
while running:
    # 绘制大型敌机
    for each in big_enemies:
```

```
        if each.active:
            each.move()
            if switch_image:
                screen.blit(each.image1, each.rect)
            else:
                screen.blit(each.image2, each.rect)
            # 即将出现在画面中,播放音效
            if each.rect.bottom == -50:
                enemy3_fly_sound.play(-1)
        else:
            # 毁灭
            if not(delay % 3):
                if e3_destroy_index == 0:
                    enemy3_down_sound.play()
                screen.blit(each.destroy_images[e3_destroy_index],\
                each.rect)
                e3_destroy_index = (e3_destroy_index + 1) % 6
                if e3_destroy_index == 0:
                    enemy3_fly_sound.stop()
                    each.reset()
...
```

## 16.13.11 发射子弹

现在的情况是我方飞机处于落后挨打的状态，敌强我弱，所以应该拿起武器进行反击！接下来定义子弹，子弹分为两种：一种是普通子弹——一次只发射一颗；另一种是补给发放的超级子弹——一次可以发射两颗。

如图 16-44 和图 16-45 所示。

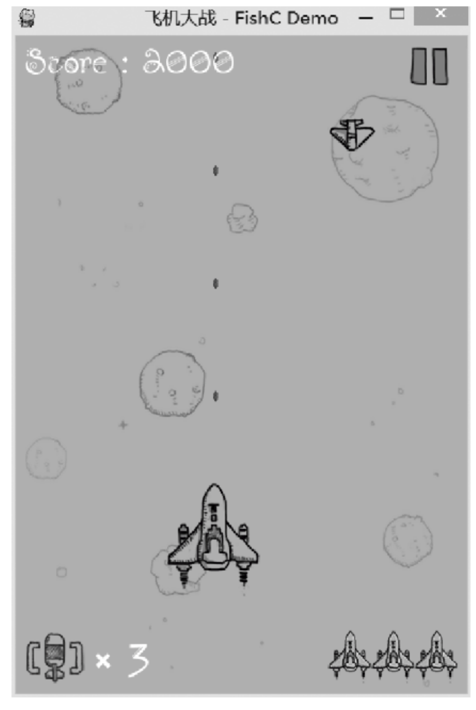



图 16-44 发射普通子弹

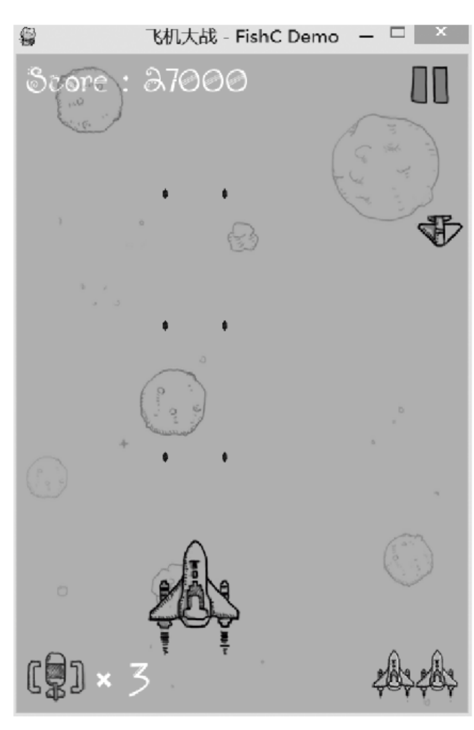



图 16-45 发射超级子弹

子弹的运动路径是直线向上，速度需要略快于飞机的速度（比飞机速度还慢的子弹总好像有哪里不对劲）。子弹移动到屏幕的尽头或击中敌机则重新绘制，因此为它添加一个 active 属性，通过该属性判断子弹是否需要重新绘制。子弹也单独定义为一个模块：

```
# bullet.py
import pygame

class Bullet1(pygame.sprite.Sprite):
    def __init__(self, position):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load("images/bullet1.png").convert_alpha()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = position
        self.speed = 12
        self.active = True
        self.mask = pygame.mask.from_surface(self.image)

    def move(self):
        self.rect.top -= self.speed
        if self.rect.top < 0:
            self.active = False

    def reset(self):
        self.rect.left, self.rect.top = position
        self.active = True
```

在 main 模块中生成子弹：

```
# main.py
…
def main():
    …
    # 生成子弹
    bullet1 = []
    bullet1_index = 0
    BULLET1_NUM = 4
    for i in range(BULLET1_NUM):
        bullet1.append(bullet.Bullet1(me.rect.midtop))
…
```

设置每 10 帧发射一颗子弹：

```
…
while running:
    # 发射子弹，每隔 10 帧射出一发
    if not(delay % 10):
        bullet1[bullet1_index].reset(me.rect.midtop)
        bullet1_index = (bullet1_index + 1) % BULLET1_NUM
…
```

接着需要检测每颗子弹是否击中敌机，并根据 active 属性判断是否绘制子弹到屏幕上：

```
…
```

```
# 检测子弹是否击中敌机
for b in bullet1:
    if b.active:
        b.move()
        screen.blit(b.image, b.rect)
        enemy_hit = pygame.sprite.spritecollide(\
        b, enemies, False, pygame.sprite.collide_mask)
        if enemy_hit:
            b.active = False
            for e in enemy_hit:
                e.active = False
…
```

程序实现如图 16-46 所示。

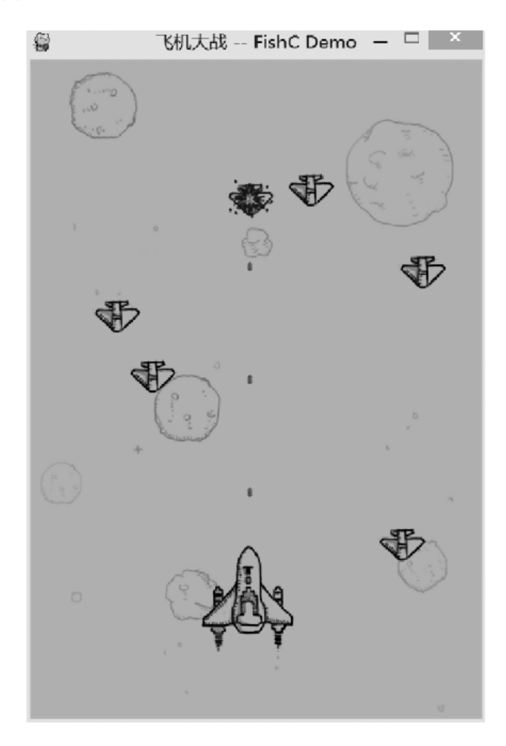



图 16-46　发射子弹

## 16.13.12　设置敌机“血槽”

敌机也不能太脆弱，对于中型和大型敌机，应该给它添加一个 energy 属性：

```
# enemy.py
class MidEnemy(pygame.sprite.Sprite):
energy = 8
    def __init__(self, bg_size):
        …
        self.energy = MidEnemy.energy
        …
    def reset(self):
        …
```

```
        self.energy = MidEnemy.energy
        …

class BigEnemy(pygame.sprite.Sprite):
energy = 20
    def __init__(self, bg_size):
        …
        self.energy = BigEnemy.energy
        …
    def reset(self):
        …
        self.energy = BigEnemy.energy
        …
```

每当中、大型敌机被子弹击中，先将 energy 属性的值减 1，直到 energy 的值为 0 才让该敌机毁灭：

```
# main.py
…
for b in bullet1:
    if b.active:
        b.move()
        screen.blit(b.image, b.rect)
        enemy_hit = pygame.sprite.spritecollide(\
        b, enemies, False, pygame.sprite.collide_mask)
        if enemy_hit:
            b.active = False
            for e in enemy_hit:
                if e in mid_enemies or e in big_enemies:
                    e.energy -= 1
                    if e.energy == 0:
                        e.active = False
                else:
                    e.active = False
…
```

可以为中、大型敌机增加一个“血槽”显示功能，这样可以更直观地让玩家知道敌机还剩下多少生命：

```
# main.py
…
# 绘制血槽
pygame.draw.line(screen, BLACK, \
                 (each.rect.left, each.rect.top - 5), \
                 (each.rect.right, each.rect.top - 5), \
                 2)
# 生命大于 20% 显示绿色，否则显示红色
energy_remain = each.energy / enemy.BigEnemy.energy
if energy_remain > 0.2:
    energy_color = GREEN
else:
    energy_color = RED
```

```
pygame.draw.line(screen, energy_color, \
                 (each.rect.left, each.rect.top - 5), \
                 (each.rect.left + each.rect.width * energy_remain, \
                  each.rect.top - 5), 2)
…
```

### 16.13.13 中弹效果

当中、大型敌机被子弹击中但并不至于毁灭的时候，应该是有“特效”的。先在 enemy.py 模块为 MidEnemy 和 BigEnemy 类添加 image_hit 属性，用于存放敌机被击中的图片。还需要一个 hit 属性，用于判断是否被子弹击中。

```
# enemy.py
class MidEnemy(pygame.sprite.Sprite):
def __init__(self, bg_size):
        …
        self.image_hit = pygame.image.load("images/enemy2_hit.png").convert_alpha()
        self.hit = False
…
class BigEnemy(pygame.sprite.Sprite):
def __init__(self, bg_size):
        …
        self.image_hit = pygame.image.load(\
        "images/enemy3_hit.png").convert_alpha()
        self.hit = False
…
```

在检测到子弹击中敌机时将对应的 hit 属性改为 True，最后绘制敌机时先检测 hit 属性，如果为 True 则绘制被击中的图片：

```
…
# 绘制大型敌机
for each in big_enemies:
    if each.active:
        …
        if each.hit:
            screen.blit(each.image_hit, each.rect)
            each.hit = False
        else:
            if switch_image:
                screen.blit(each.image1, each.rect)
            else:
                screen.blit(each.image2, each.rect)
…
# 绘制中型敌机：
for each in mid_enemies:
    if each.active:
        …
        if each.hit:
            screen.blit(each.image_hit, each.rect)
            each.hit = False
```

```
            else:
                screen.blit(each.image, each.rect)
    …
```

## 16.13.14 绘制得分

游戏界面的左上角应该显示玩家的得分并实时更新，击中小、中、大型敌机分别可以获得1000分、6000分和10000分。有些读者朋友可能会觉得1000分作为基本单位显得有点浮夸，不过这完全是游戏开发的业界习惯。

增加一个score变量用于记录玩家得分，当敌机被消灭的时候，加上对应的分数：

```
…
def main():
    …
    score = 0
    score_font = pygame.font.Font("font/font.TTF", 36)
    …
    while running:
        # 大、中、小敌机在毁灭时，score 分别增加 10000、6000 和 1000 分
        …
        # 绘制得分
        score_text = score_font.render(\
        "Score : %s" % str(score), True, WHITE)
        screen.blit(score_text, (10, 5))
…
```

程序实现如图16-47所示。

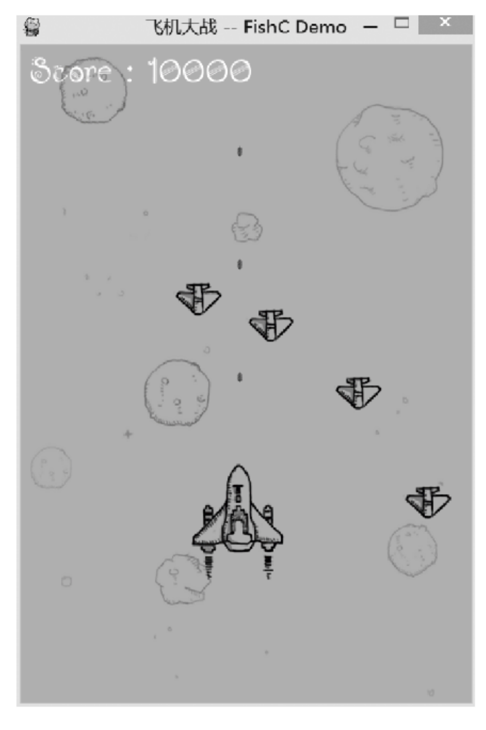



图16-47　绘制得分

### 16.13.15 暂停游戏

右上角可以添加一个暂停按钮，让玩家随时可以把游戏暂停下来。暂停按钮总共有四种样式，如图 16-48 所示。



图 16-48 暂停按钮

这些按钮分别代表继续游戏和暂停游戏的命令，其中深色的图标表示鼠标停留在按钮上方时显示的样式。通过响应 MOUSEBUTTONDOWN 事件并判断鼠标的位置可以得知玩家是否按下了暂停按钮，通过响应 MOUSEMOTION 事件修改暂停按钮的样式：

```
# main.py
...
def main():
    ...
    # 是否暂停游戏
    paused = False
    pause_nor_image = \
    pygame.image.load("images/pause_nor.png").convert_alpha()
    pause_pressed_image = \
    pygame.image.load("images/pause_pressed.png").convert_alpha()
    resume_nor_image = \
    pygame.image.load("images/resume_nor.png").convert_alpha()
    resume_pressed_image = \
    pygame.image.load("images/resume_pressed.png").convert_alpha()
    paused_rect = pause_nor_image.get_rect()
    paused_rect.left, paused_rect.top = \
    width - paused_rect.width - 10, 10
    # 默认显示这个
    paused_image = pause_nor_image
    ...
    while running:
        for event in pygame.event.get():
            ...
            elif event.type == MOUSEBUTTONDOWN:
                if event.button == 1 and \
                paused_rect.collidepoint(event.pos):
                    paused = not paused
            elif event.type == MOUSEMOTION:
                if paused_rect.collidepoint(event.pos):
                    if paused:
                        paused_image = resume_pressed_image
                    else:
                        paused_image = pause_pressed_image
                else:
                    if paused:
                        paused_image = resume_nor_image
                    else:
                        paused_image = pause_nor_image
...
```

接着让游戏的主流程只有在 paused 为 False 的时候才得以执行，另外还需要将 screen.blit(background, (0, 0))提取出来，这样玩家就没办法通过不断地暂停、继续游戏来实现"作弊"的行为。

```
# main.py
…
while running:
    # 事件循环
    screen.blit(background, (0, 0))
    if not paused:
        # 游戏主流程
    # 绘制暂停按钮
    screen.blit(paused_image, paused_rect)
…
```

### 16.13.16 控制难度

敌人的速度如果一成不变（一直维持慢悠悠的移动），那么对于玩家来说是无法接受的。因为玩家希望得到的游戏体验是刺激，是心跳！所以要让游戏的难度随着得分的增加而增加。这里将游戏划分为 5 个级别，每提升一个级别，就增加一些敌机，或提高敌机的移动速度。

```
…
def inc_speed(target, inc):
    for each in target:
        each.speed += inc

def main():
    …
    # 设置难度级别
    level = 1
    …
    while running:
        …
        # 根据用户分数增加难度
        if level == 1 and score > 50000:
            level = 2
            upgrade_sound.play()
            # 增加 3 架小型敌机、2 架中型敌机和 1 架大型敌机
            add_small_enemies(small_enemies, enemies, 3)
            add_mid_enemies(mid_enemies, enemies, 2)
            add_big_enemies(big_enemies, enemies, 1)
            # 提升小型敌机的速度
            inc_speed(small_enemies, 1)
        elif level == 2 and score > 300000:
            level = 3
            upgrade_sound.play()
            # 增加 5 架小型敌机、3 架中型敌机和 2 架大型敌机
            add_small_enemies(small_enemies, enemies, 5)
            add_mid_enemies(mid_enemies, enemies, 3)
            add_big_enemies(big_enemies, enemies, 2)
```

```
            # 提升小、中型敌机的速度
            inc_speed(small_enemies, 1)
            inc_speed(mid_enemies, 1)
        elif level == 3 and score > 600000:
            level = 4
            upgrade_sound.play()
            # 增加 5 架小型敌机、3 架中型敌机和 2 架大型敌机
            add_small_enemies(small_enemies, enemies, 5)
            add_mid_enemies(mid_enemies, enemies, 3)
            add_big_enemies(big_enemies, enemies, 2)
            # 提升小、中型敌机的速度
            inc_speed(small_enemies, 1)
            inc_speed(mid_enemies, 1)
        elif level == 4 and score > 1000000:
            level = 5
            upgrade_sound.play()
            # 增加 5 架小型敌机、3 架中型敌机和 2 架大型敌机
            add_small_enemies(small_enemies, enemies, 5)
            add_mid_enemies(mid_enemies, enemies, 3)
            add_big_enemies(big_enemies, enemies, 2)
            # 提升小、中型敌机的速度
            inc_speed(small_enemies, 1)
            inc_speed(mid_enemies, 1)
...
```

### 16.13.17 全屏炸弹

其实只要到了 level5 的时候，就会下飞机雨，这时玩家就很容易陷入不利的局面。因此，游戏为玩家提供了全屏炸弹这一超级杀招。此招一出，界面上所有的敌机将会在一瞬间灰飞烟灭，让玩家谈笑于千里之外。

通过空格键可以触发全屏炸弹，初始情况下有三颗全屏炸弹，可以通过补给获得，但最多只能装载三颗。由于触发全屏炸弹是属于偶然的操作，因此通过响应 KEYDOWN 事件再检测用于 event.key 是否为 K_SPACE 即可：

```
...
def main():
    ...
    # 全屏炸弹
    bomb_image = pygame.image.load("images/bomb.png").convert_alpha()
    bomb_rect = bomb_image.get_rect()
    bomb_font = pygame.font.Font("font/font.ttf", 48)
    bomb_num = 3
    ...
    while running:
        for event in pygame.event.get():
            ...
            elif event.type == KEYDOWN:
                if event.key == K_SPACE:
                    if bomb_num:
                        bomb_num -= 1
```

```
                    bomb_sound.play()
                    for each in enemies:
                        if each.rect.bottom > 0:
                            each.active = False
            …
            # 绘制全屏炸弹数量
            bomb_text = bomb_font.render("× %d" % bomb_num, True, WHITE)
            text_rect = bomb_text.get_rect()
            screen.blit(bomb_image, (10, height - 10 - bomb_rect.height))
            screen.blit(bomb_text, \
             (20 + bomb_rect.width, height - 5 - text_rect.height))
…
```

## 16.13.18 发放补给包



游戏设计每 30 秒随机发放一个补给包，可能是超级子弹，也可能是全屏炸弹。补给包有自己的图像和运动轨迹，不妨单独为其定义一个模块：

```
# supply.py
import pygame
from random import *

class Bullet_Supply(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load( \
        "images/bullet_supply.png").convert_alpha()
        self.rect = self.image.get_rect()
        self.width, self.height = bg_size[0], bg_size[1]
        self.rect.left, self.rect.bottom = randint(
        0, self.width - self.rect.width), -100
        self.speed = 5
        self.active = False
        self.mask = pygame.mask.from_surface(self.image)

    def move(self):
        if self.rect.top < self.height:
            self.rect.top += self.speed
        else:
            self.active = False

    def reset(self):
        self.active = True
        self.rect.left, self.rect.bottom = randint(\
        0, self.width - self.rect.width), -100

class Bomb_Supply(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load("images/bomb_supply.png")
        self.rect = self.image.get_rect()
```

```
        self.width, self.height = bg_size[0], bg_size[1]
        self.rect.left, self.rect.bottom = randint(\
        0, self.width - self.rect.width), - 100
        self.speed = 5
        self.active = False
        self.mask = pygame.mask.from_surface(self.image)

    def move(self):
        if self.rect.top < self.height:
            self.rect.top += self.speed
        else:
            self.active = False

    def reset(self):
        self.rect.left, self.rect.bottom = randint(\
        0, self.width - self.rect.width), - 100
        self.active = True
```

在 main 模块中实例化补给包，并设置一个补给包发放定时器，每三十秒随机发放一个补给包：

```
# main.py
…
def main():
    …
    # 每 30 秒发一个补给包
    bomb_supply = supply.Bomb_Supply(bg_size)
    bullet_supply = supply.Bullet_Supply(bg_size)
    SUPPLY_TIME = USEREVENT
    pygame.time.set_timer(SUPPLY_TIME, 30 * 1000)
    …
    while running:
        for event in pygame.event.get():
            …
            elif event.type == SUPPLY_TIME:
                supply_sound.play()
                if choice([True, False]):
                    bomb_supply.reset()
                else:
                    bullet_supply.reset()
        …
        if not paused:
            …
            # 绘制全屏炸弹补给并检测是否获得
            if bomb_supply.active:
                bomb_supply.move()
                screen.blit(bomb_supply.image, bomb_supply.rect)
                if pygame.sprite.collide_mask(bomb_supply, me):
                    get_bomb_sound.play()
                    if bomb_num < 3:
                        bomb_num += 1
                    bomb_supply.active = False

            # 绘制超级子弹补给并检测是否获得
```

```
if bullet_supply.active:
    bullet_supply.move()
    screen.blit(bullet_supply.image, bullet_supply.rect)
    if pygame.sprite.collide_mask(bullet_supply, me):
        get_bullet_sound.play()
        # 发射超级子弹
        bullet_supply.active = False
```

程序实现如图 16-49 所示。

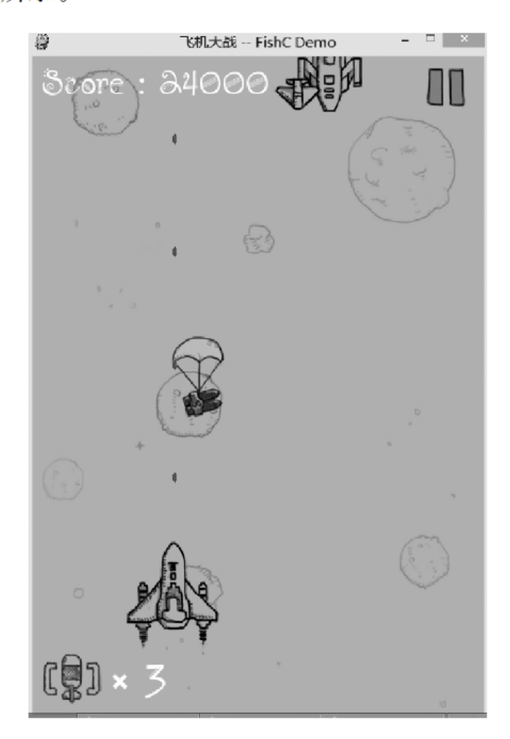



图 16-49　发放补给包

接下来有个细节问题，就是当用于单击暂停按钮的时候，补给计时器应该暂停，否则每隔一段时间就会听到发放补给的声音。另外，背景音乐和其他音效也应该暂停，因为玩家既然单击了暂停按钮，可能是要接个电话或者出去打个酱油，所以程序还是安静地等着就可以了：

```
…
elif event.type == MOUSEBUTTONDOWN:
    if event.button == 1 and paused_rect.collidepoint(event.pos):
        paused = not paused
        # 暂停时停止补给发放和背景音乐
        if paused:
            pygame.time.set_timer(SUPPLY_TIME, 0)
            pygame.mixer.music.pause()
            pygame.mixer.pause()
        else:
            pygame.time.set_timer(SUPPLY_TIME, 30 * 1000)
            pygame.mixer.music.unpause()
            pygame.mixer.unpause()
…
```

### 16.13.19 超级子弹

当接到超级子弹补给包的时候，子弹由原先的一次发射一发变成两发，子弹的速度也相对会快一些。先在 bullet 模块中添加 Bullet2 类来描述超级子弹：

```
# bullet.py
class Bullet2:
    def __init__(self, position):
        pygame.sprite.Sprite.__init__(self)
        self.image = \
        pygame.image.load("images/bullet2.png").convert_alpha()
        self.rect = self.image.get_rect()
        self.rect.left, self.rect.top = position
        self.speed = 14
        self.active = True
        self.mask = pygame.mask.from_surface(self.image)

    def move(self):
        self.rect.top -= self.speed
        if self.rect.top < 0:
            self.active = False

    def reset(self, position):
        self.rect.left, self.rect.top = position
        self.active = True
```

超级子弹所向披靡，所以要限制使用时间为 18 秒，过了这个时间就自动变回普通子弹。因此需要一个超级子弹定时器，还需要用一个变量来表示子弹的发射类型。

```
# main.py
…
def main():
    …
    # 生成超级子弹
    bullet2 = []
    bullet2_index = 0
    BULLET2_NUM = 8
    for i in range(BULLET2_NUM//2):
        bullet2.append(bullet.Bullet2( \
        (me.rect.centerx - 33, me.rect.centery)))
        bullet2.append(bullet.Bullet2( \
        (me.rect.centerx + 30, me.rect.centery)))
    …
    # 超级子弹定时器
    DOUBLE_BULLET_TIME = USEREVENT + 1
    # 是否使用超级子弹
    is_double_bullet = False
    …
    while running:
        for event in pygame.event.get():
            …
```

```
            elif event.type == DOUBLE_BULLET_TIME:
                is_double_bullet = False
                pygame.time.set_timer(DOUBLE_BULLET_TIME, 0)
        …
        if not paused:
            …
            # 绘制超级子弹补给并检测是否获得
            if bullet_supply.active:
                bullet_supply.move()
                screen.blit(bullet_supply.image, bullet_supply.rect)
                if pygame.sprite.collide_mask(bullet_supply, me):
                    get_bullet_sound.play()
                    is_double_bullet = True
                    # 超级子弹限制使用 18 秒
                    pygame.time.set_timer(DOUBLE_BULLET_TIME, 18 * 1000)
                    bullet_supply.active = False
            # 发射子弹
            if not(delay % 10):
                if is_double_bullet:
                    bullets = bullet2
                    bullets[bullet2_index].reset( \
                    (me.rect.centerx - 33, me.rect.centery))
                    bullets[bullet2_index + 1].reset( \
                    (me.rect.centerx + 30, me.rect.centery))
                    bullet2_index = (bullet2_index + 2) % BULLET2_NUM
                else:
                    bullets = bullet1
                    bullets[bullet1_index].reset(me.rect.midtop)
                    bullet1_index = (bullet1_index + 1) % BULLET1_NUM
                bullet_sound.play()
            # 检测子弹是否击中敌机
            for b in bullets:
                …
…
```

## 16.13.20 三次机会

很多游戏都会给玩家多次尝试的机会，因此也会添加这么一个功能。玩家总共会有 3 次机会，游戏界面右下角的小飞机代表还有多少次机会，如图 16-50 所示。

现在 myplane 模块中添加一个 reset()方法，用于重新诞生一个新的飞机：

```
# myplane.py
…
def reset(self):
    self.rect.left, self.rect.top = \
    (self.width - self.rect.width) // 2, \
    self.height - self.rect.height - 60
    self.active = True
…
```

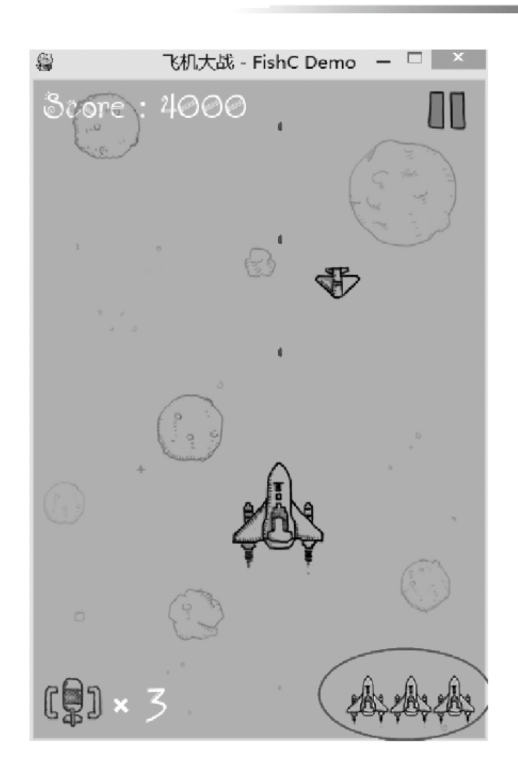



图 16-50　提供多次机会

接着修改 main 模块，增加一个 life_num＝3 变量，在我方飞机毁灭时 life_num 减 1，并在界面的右下角显示还有多少次机会：

```
# main.py
...
def main():
    ...
    # 生命数量
    life_image = pygame.image.load("images/life.png").convert_alpha()
    life_rect = life_image.get_rect()
    life_num = 3
    ...
    while running:
        if life_num and not paused:
            ...
            # 绘制我方飞机
            if me.active:
                if switch_image:
                    screen.blit(me.image1, me.rect)
                else:
                    screen.blit(me.image2, me.rect)
            else:
                # 毁灭
                if not(delay % 3):
                    if me_destroy_index == 0:
                        me_down_sound.play()
                    screen.blit( \
```

```
                    me.destroy_images[me_destroy_index], me.rect)
                    me_destroy_index = (me_destroy_index + 1) %\ 4
                    if me_destroy_index == 0:
                        life_num -= 1
                        me.reset()
            …
            # 绘制剩余生命的数量
            if life_num:
                for i in range(life_num):
                    screen.blit(life_image, \
                    (width - 10 - (i + 1) * life_rect.width, \
                    height - 10 - life_rect.height))
        # 游戏结束画面
        elif life_num == 0:
            print("Game Over!")
…
```

这里有个小细节，就是每次我方飞机牺牲后，如果诞生的位置刚好有敌机，那么会导致我方飞机一诞生就牺牲的惨剧。因此可以设定每次牺牲后会有 3 秒钟的安全期，在安全期内敌机是无法伤害到你的。

具体做法就是在 Myplane 中加入一个 invincible 属性，该属性为 True 时我方飞机处于一个无敌状态：

```
# myplane.py
…
class MyPlane(pygame.sprite.Sprite):
    def __init__(self, bg_size):
        …
        self.invincible = False
…
    def reset(self):
        …
        self.invincible = True
…
```

新飞机诞生时，设置一个 3 秒钟的定时器：

```
# main.py
…
def main():
    …
    # 解除我方无敌状态
    INVINCIBLE_TIME = USEREVENT + 2
    …
    while running:
        for event in pygame.event.get():
            …
            elif event.type == INVINCIBLE_TIME:
                me.invincible = False
                pygame.time.set_timer(INVINCIBLE_TIME, 0)
        …
        if life_num and not paused:
            …
```

```
# 检测我方飞机是否被撞
enemies_down = pygame.sprite.spritecollide( \
me, enemies, False, pygame.sprite.collide_mask)
if enemies_down and not me.invincible:
    me.active = False
    for e in enemies_down:
        e.active = False
# 绘制我方飞机
if me.active:
    if switch_image:
        screen.blit(me.image1, me.rect)
    else:
        screen.blit(me.image2, me.rect)
else:
    # 毁灭
    if not(delay % 3):
        if me_destroy_index == 0:
            me_down_sound.play()
        screen.blit(me.destroy_images[me_destroy_index], \
        me.rect)
        me_destroy_index = (me_destroy_index + 1) % 4
        if me_destroy_index == 0:
            life_num -= 1
            me.reset()
            pygame.time.set_timer(INVINCIBLE_TIME, 3 * 1000)
```

## 16.13.21 结束画面

当 life_num 的值为 0 时，说明玩家已经输掉了游戏，进入游戏结束画面，如图 16-51 所示。




图 16-51　游戏结束画面

游戏结束时，结束画面会显示历史最高得分，以及玩家的最终成绩。如果玩家的最终成绩比历史最高得分要高，那么将玩家成绩写入存档。另外，结束画面有“重新开始”和“结束游戏”两个按钮：

```
# main.py
…
def main():
    …
    # 用于阻止重复打开记录文件
    recorded = False
    # 游戏结束画面
    gameover_font = pygame.font.Font("font/font.TTF", 48)
    again_image = pygame.image.load("images/again.png").convert_alpha()
    again_rect = again_image.get_rect()
    gameover_image = \
    pygame.image.load("images/gameover.png").convert_alpha()
    gameover_rect = gameover_image.get_rect()
    …
    while running:
        …
        if life_num and not paused:
            …
        # 游戏结束画面
        elif life_num == 0:
            # 背景音乐停止
            pygame.mixer.music.stop()
            # 停止全部音效
            pygame.mixer.stop()
            # 停止补给发放
            pygame.time.set_timer(SUPPLY_TIME, 0)
            if not recorded:
                recorded = True
                # 读取历史最高得分
                with open("record.txt", "r") as f:
                    record_score = int(f.read())
                # 如果玩家得分高于历史最高得分,则存档
                if score > record_score:
                    with open("record.txt", "w") as f:
                        f.write(str(score))
            # 绘制结束画面
            record_score_text = score_font.render( \
            "Best : %d" % record_score, True, (255, 255, 255))
            screen.blit(record_score_text, (50, 50))
            gameover_text1 = gameover_font.render( \
            "Your Score", True, (255, 255, 255))
            gameover_text1_rect = gameover_text1.get_rect()
            gameover_text1_rect.left, gameover_text1_rect.top = \
            (width - gameover_text1_rect.width) // 2, height // 3
            screen.blit(gameover_text1, gameover_text1_rect)
            gameover_text2 = gameover_font.render( \
            str(score), True, (255, 255, 255))
```

```
gameover_text2_rect = gameover_text2.get_rect()
gameover_text2_rect.left, gameover_text2_rect.top =\
(width - gameover_text2_rect.width) // 2,\
gameover_text1_rect.bottom + 10
screen.blit(gameover_text2, gameover_text2_rect)
again_rect.left, again_rect.top = \
(width - again_rect.width) // 2, \
gameover_text2_rect.bottom + 50
screen.blit(again_image, again_rect)
gameover_rect.left, gameover_rect.top = \
(width - again_rect.width) // 2, \
again_rect.bottom + 10
screen.blit(gameover_image, gameover_rect)
# 检测用户的鼠标操作
# 如果用户按下鼠标左键
if pygame.mouse.get_pressed()[0]:
    # 获取鼠标坐标
    pos = pygame.mouse.get_pos()
    # 如果用户单击"重新开始"
    if again_rect.left < pos[0] < again_rect.right \
    and again_rect.top < pos[1] < again_rect.bottom:
        # 调用 main 函数,重新开始游戏
        main()
    # 如果用户单击"结束游戏"
    elif gameover_rect.left < pos[0] < \
    gameover_rect.right and gameover_rect.top < pos[1] < \
    gameover_rect.bottom:
        # 退出游戏
        pygame.quit()
        sys.exit()
...
```

# 参 考 文 献


［1］ Wesley J. Chun. Python 核心编程［M］. 2 版. 宋吉广，译. 北京：人民邮电出版社，2008.

［2］ Magnus Lie Hetland. Python 基础教程［M］. 2 版. 司维，曾军崴，谭颖华，译. 北京：人民邮电出版社，2010.

［3］ Warren Sande，Carter Sande. 与孩子一起学编程［M］. 苏金国，等译. 北京：人民邮电出版社，2010.

［4］ Alex Martelli，Anna Ravenscroft，David Ascher. Python Cookbook（中文版）［M］. 高铁军，译. 北京：人民邮电出版社，2010.

［5］ Mark Pilgrim. Dive Into Python 3. Apress，2009.